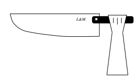
UNIVERSITEIT
GENT

DEPARTMENT OF APPLIED MATHEMATICS,
BIOMETRICS AND PROCESS CONTROL

PSYCO

PROTOTYPE SYMBOLIC CONTROL TOOLBOX



Gian Lorenzo Lucchetti
Marco Zoccadelli

Supervisor: Hans Vangheluwe

September 21, 1998

Contents

Introduction

PSYCO (Prototype SYmbolic COntrol toolbox) is software which allows one to analyze a dynamic system from a control engineering point of view. It was born as a collaboration between the DSI (Dipartimento Sistemi e Informatica) of the faculty of engineering of Florence University and the BIOMATH department at the agricultural faculty of Gent University, within the framework of a Socrates/Erasmus student exchange program.
PSYCO is related to a wider project called WEST++. WEST++ is a software for modeling and simulation of dynamic systems, in particular Waste Water Treatment Plants (WWTPs) and has been developed at the BIOMATH department WEST++ is a powerful software to build models and simulate them, but so far no control facilities were available. The purpose of PSYCO has been to give the opportunity to a user to analyze models through classical control tools and then design controllers to impose a desired system behavior.
Both PSYCO and WEST++ are based on the same philosophy, that is to solve problems in a symbolic fashion as much as possible; that means they seek for an analytical solution of a problem, whenever it exists, or use symbolic manipulations to improve performance of subsequent numerical approaches.
This report is composed of three parts:

- Background
- PSYCO
- Aeration Tank

The "Background" part tries to explain all the knowledge which has been necessary to build the PSYCO software; that means theoretical knowledge (symbolic manipulation, Waste Water Treatment Plants (WWTPs), modeling and simulation) and the corresponding computer implementation (computer algebra systems, WEST++ modeling and experimental environment).
Part two (PSYCO) is a detailed description of the Prototype Symbolic Control Toolbox. Each chapter starts from issues in System Theory and Automatic Control and explains how it has been possible to realize them in a symbolic manipulation environment.
As a result of the strong orientation of BIOMATH towards WWTP applications, PSYCO has been applied to a simplified IAWQ [6] model of n.1 and the results are shown in the third part of this report.
Figure 1 shows what can be done by means of PSYCO. The starting point is a model of a plant, either in state equation form or in transfer function form. Then, a menu driven interface allows the user to reach all the blocks depicted in the figure. The final goal is to assist in the design of controllers for the plant.
Once a controller has been designed, a complete controlled system (*i.e.,* plant, controller and their interconnections) can be simulated by means of WEST++. From a user point of view, that is an easy task: it is sufficient to choose and connect blocks, and automatically MSL-EXEC code (Model Specification Language Exec level, basically C++ code) is available for the simulation. Actually, our work to link PSYCO and WEST++ was not trivial. In fact, although libraries supporting WWTP models were already present, control libraries were not and a further programming effort in MSL-USER (Model Specification Language, User level) has necessary. Figure 2 depicts links between PSYCO and WEST++. Hereby, a Roman font denotes static entities (such as a State Space model), whereas an Italic font denotes transformation activities.

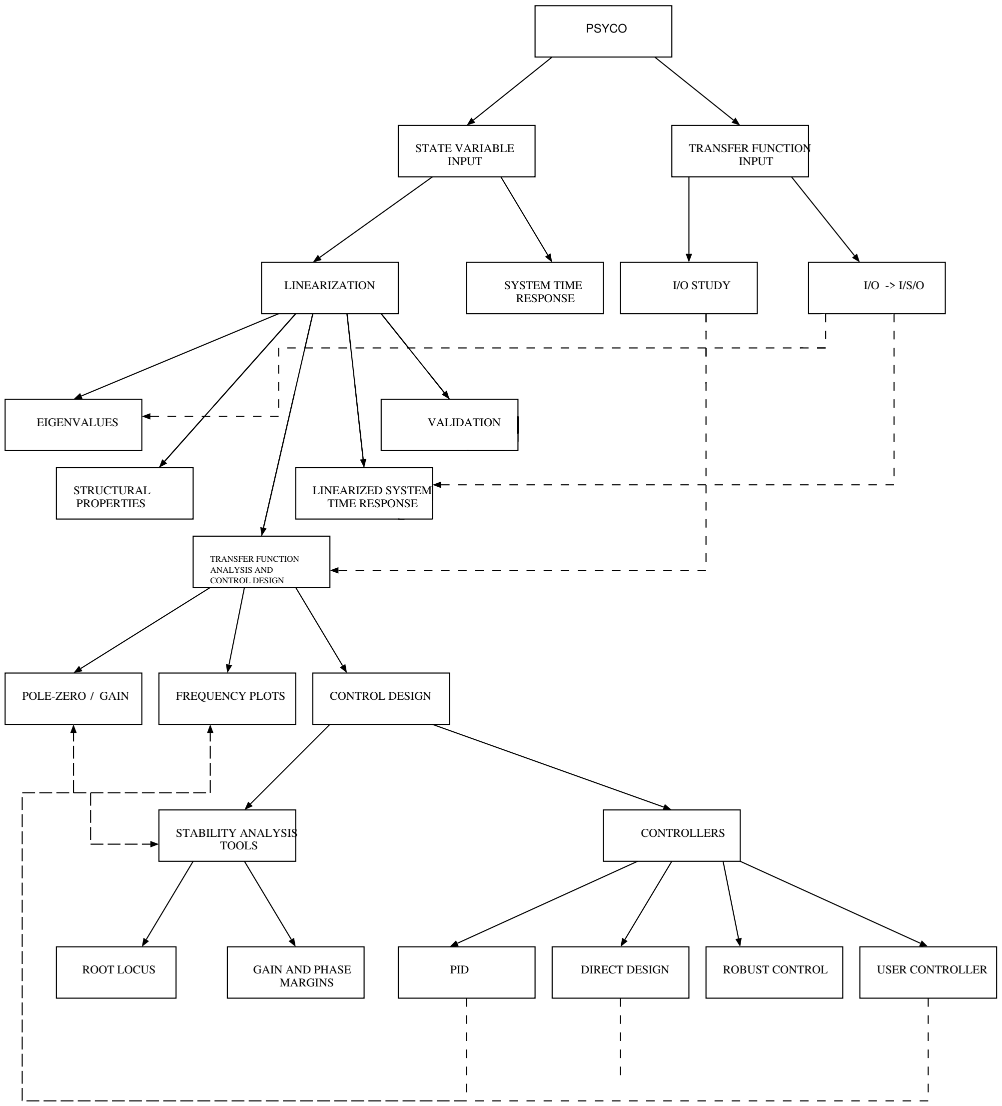


Figure 1: PSYCO

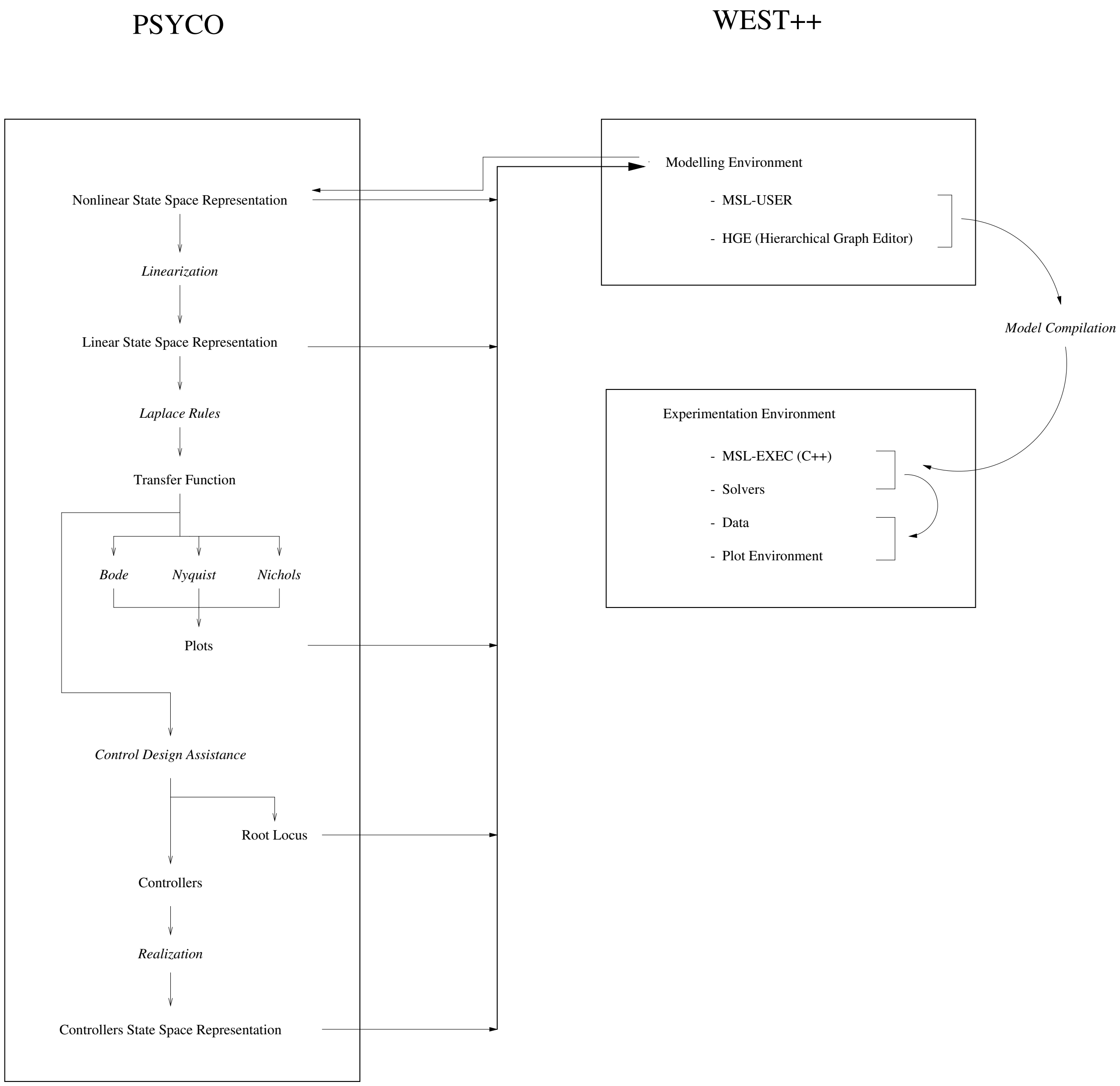


Figure 2: Links between PSYCO and WEST++

Part I

BACKGROUND

1 Symbolic Manipulation

Symbolic Manipulation is concerned with finding symbolic or exact solutions to mathematical problems. This avoids rounding errors and the need for an error analysis. From a symbolic solution one may get more insight into a problem and derive whole classes of solutions. Exact or symbolic computation has the disadvantage to be more compute-intensive than numerical calculation. However, as symbolic manipulation is usually performed only once as opposed to numerical code which gets executed time and again during simulation, the one-time intensive symbolic computation cost is largely compensated by the performance gain at simulation time.

1.1 Scientific Calculation and Algebraic Calculation

From the start of electronic calculation, one of the main uses of computers has been numerical calculation. Very soon, applications to management began to dominate the scene, as far as the volume of calculation assigned to them is concerned. Nevertheless, scientific applications are still the most prestigious, especially if we look at the performance required of the computer: the most powerful computers are usually reserved for scientific calculation.

This concept of "scientific calculation" conceals an ambiguity, which it is important to note: before computers appeared on the scene, a calculation usually consisted of a mixture of numerical calculation and what we shall call "algebraic calculation", that is calculation by mathematical formulae. The only example of purely numerical calculation seems to have been the feats of calculating prodigies such as Inaudi: the authors of tables, especially of logarithms, did indeed carry out enormous numerical calculations, but these were preceded by a restatement of the algebraic formulae and methods which were essential if the work was to be within the bounds of what is humanly possible. For example, the famous large calculation of the 19th century include a large proportion of formula manipulation. The best known is certainly Le Verrier's calculation of the orbit of Neptune, which started from the disturbances of the orbit of Uranus, and which led to the discovery of Neptune. The most impressive calculation with pencil and paper is also in the field of astronomy: Delaunay took 10 years to calculate the orbit of the moon, and another 10 years to check it. The result is not numerical, because it consists for the most part of a formula which by itself occupies all the 128 pages of Chapter 4 of his book.

The ambiguity mentioned above is the following: when computers came on the scene, numerical calculation was made very much easier and it became commonplace to do enormous calculations, which in some cases made it possible to avoid laborious algebraic manipulation. The result was that, for the public at large and even for most scientists, numerical calculation and scientific calculation have become synonymous. However, numerical calculation does not rule out algebraic calculation: writing the most trivial numerical program requires a restatement of the formulae on which the algorithm is based.

Thus algebraic calculation has not lost its relevance. However, it is most frequently done with pencil and paper, even though the first software intended to automate it is already quite old. It was very quickly seen that such software for helping algebraic calculation would have to be a complete system, which included a method with a very special structure for representing non numerical data, a language making it possible to manipulate them, and a library of effective functions for carrying out the necessary basic algebraic operations.
This discipline is known as “Symbolic Manipulation”, or “Algebraic Calculation” or “Computer Algebra”.
The most intuitive, although somewhat restrictive, approach to Computer Algebra systems is to say that they are made for the manipulation of everyday scientific and engineering formulae. A mathematical formula which is described in one of the usual languages (FORTRAN, PASCAL, C, BASIC,...) can only be evaluated numerically once the variables and parameters have themselves been given numerical values. In a language which allows algebraic manipulation, the same formula can also be evaluated numerically, but above all it can be the object of formal transformation: differentiation, development in series, various expansions, integration, etc...
The areas typically covered by computer algebra systems are:

- Operations on integers, on rational, real and complex numbers with unlimited accuracy.
- Operations on polynomials in one or more variables and on rational fractions. In short, the obvious rational operations, calculating the g.c.d., factorising over the integers, ...
- Calculation on matrices with numerical and/or symbolic elements.
- Simple analysis: differentiation, expansion in series, ...
- Manipulation of formulae: various substitutions, selection of coefficient and parts of formulae, numerical evaluation, pattern recognition, controlled simplification, ...

Starting out from this common base, the system may offer possibilities in specific areas, possibilities which to some extent characterize their degree of development. For example:

- Solution of equations.
- Formal integration.
- Calculation of limits.
- Tensor calculus.

1.2 Symbolic Manipulation: Why

Imagine you are trying to solve an equation for an unknown variable, such as: x - 5 = 0. We say we have an analytic solution if we can actually solve the equation explicitly for the unknown variable. In this case, it is easy to see that the explicit analytic solution is x = 5, and that this is the exact solution.
If we were not so smart, we might develop an “algorithm” on a computer to solve this equation numerically. The algorithm would test various values for x, and then stop with a “solution” when the equation was satisfied to some chosen tolerance. For example, we might demand that the computer should solve this equation to an accuracy of 0.5. Then the computer would follow the algorithm until it found a solution to this degree of accuracy. Given an initial guess x = 1, depending on the algorithm, it might come up with the following guesses: x = 2.2 (no good), x = 3.3 (no good), x = 4.6 (good to the tolerance we specified), and return the “solution” x = 4.6. An efficient algorithm would come up with a solution quickly.
Note that if we want to be more accurate, as scientists do in their predictions, we might specify a tolerance that is much smaller, like 0.001. The computer might eventually get a result after the following sequence: 2.2, 3.3, 4.6, 5.2, 5.05, 4.98, 5.0005, and then return with the solution x = 5.0005. Of course, it takes much more work to solve the equation to this level of accuracy.

For such a simple equation we would never use a computer to get a solution. When the equations to be solved are large and complex, describing many coupled physical processes dynamics and containing a lot of terms, several efforts have to be done to obtain the right solution; one has to deal with some issues about how to reach it in the most efficient way and why one should prefer one way to tackle a problem rather than another. Several problems can be tackled both in a numerical and a symbolical way. Getting the solution using one method rather than the other, can have advantages and disadvantages.
The main advantages in favor of a symbolic approach are:

- Performance, if you know a quantity analytically, you can avoid some computations and decrease the computation time (for example, two identical expressions with opposite sign composed of a lot of terms, can be simplified before doing all the calculations involved, provided that expressions are known analytically);
- Re-use, if you get an analytical solution, you get a whole class of solutions (for example, if you have to compute an integral on an interval [a, b] and then on [c, d], if you have computed the analytical primitive you have just to evaluate it twice, otherwise you have to proceed with two numerical integrations, surely more CPU consuming)
- Correctness, sometimes it is impossible to obtain a correct solution numerically (*e.g.,* , if you have a linear system of equations with determinant equal zero, you have infinite solutions that can be written only symbolically)
- More accurate numerical results, because, preprocessing data with symbolical manipulations, more advanced numerical technics can be exploited (*e.g.,* , using analytical forms of derivatives for the purpose of Taylor approximations)

On the other hand, symbolic computation programs are regarded as fairly limited in most of the following areas:

- Solving non-linear ODEs and PDEs
- Solving non-linear algebraic equations
- Solving non-linear optimization problems

These problems are the province of numerical computation, because adequate analytic solution methods may not exist and therefore may not be implemented in a symbolic computation package. However symbolic methods can be used to derive expressions necessary for performing numerical computations–such as gradients and Jacobian and Hessian matrices. Thus, the traditional roles of numeric and symbolic computation are not distinct and many benefits arise from merging the two.

1.3 Symbolic Manipulation: How

Several computer algebra systems can handle symbolic manipulation (MACSYMA, REDUCE, DERIVE, MAPLE, MATHEMATICA, AXIOM). Our work has been based on MuPAD `www.mupad.de`.
MuPAD is a computer algebra system which, up to now, has been developed mainly at the University of Paderborn. MuPAD provides mathematical functions such as mentioned above. For example, if one wants to obtain the indefinite integral of

$$\frac{(x^2+2)(x-1)}{x^2(x^2+1)}$$

MuPAD gives the solution:

$$2\ln(x) + 2/x - \arctan(1/x) - \frac{\ln(x^2+1)}{2}$$

Expressions may not only be computed symbolically, one may also do numerical computation (for example evaluation of the previous integral at a certain instant). Let us have a look more detailed at the components of the MuPAD system.
MuPAD is a general purpose computer algebra system. MuPAD consists of a so-called kernel which, for speed and efficiency, is mainly implemented in the programming language C and partially written in C++. The MuPAD kernel is composed of the following basic parts:

- the arithmetic, handling numbers of arbitrary lengths.
- the parser, reading and checking the user's input.
- the evaluator, evaluating and simplifying input data.
- the memory management MAMMUT (Memory Allocation Management UniT), handling all data of the system MAMMUT uses a (weak) unique data representation and serves as an interface between the hardware and the MuPAD kernel. Apart from the graphical user interface, MAMMUT is the only platform dependent part of the MuPAD system.
- built-in functions. These are user accessible functions which are implemented in the kernel for speed and efficiency, *e.g.,* , often needed functions for manipulating arithmetical expressions or polynomials.

The MuPAD kernel features a high-level programming language, the MuPAD language. The mathematical expertise of MuPAD mainly resides in libraries written in the MuPAD language which are thus independent of the underlying system platform.
So called Dynamic Modules allow to extend the MuPAD kernel or integrate software packages within MuPAD. A dynamic module contains so-called module functions which are, in contrast to library functions, not written in the programming language but are compiled machine code functions written in C/C++. They can directly access internal kernel methods.
MuPAD offers data types (graphical primitives such as points and polygons) and functions for plotting two- and three-dimensional curves and surfaces. A two- or three-dimensional plot (a "scene") is displayed with the MuPAD graphics tool VCam. A scene consists of an object or a sequence of objects. For each object several parameters such as text, color and styles can be defined. Further parameters are the "viewing point" defining the perspective, the scale, axes and their labeling, back- and foreground color, and more. A scene can be saved in different graphic formats, or can be send to a printer. VCam even offers the possibility to generate and manipulate a scene interactively. Any such scene can be translated to a corresponding MuPAD input.
The graphical front-end of the MuPAD help tool is organized as a hypertext system. The user can navigate to the list of available functions, to the index or to the contents page of the on-line manuals, can create his own hyperlinks, set bookmarks and so on. The entire MuPAD documentation (i.e, the language description as well as the complete library documentation) is organized as hypertext documents for the MuPAD help tool.
In addition to this, MuPAD has a window based Source-Code Debugger for debugging user-defined procedures and domains which were written in the MuPAD language.
MuPAD is available for different computer platforms including various Unix systems (Linux, Sun OS, Solaris, SGI, ...), the Macintosh OS and Microsoft Windows 95/NT.

2

Modelling and Simulation

2.1 Theory of Modelling and Simulation

In the following, a short introduction to the basic concepts of modelling and simulation is given. Further, the current WEST++ implementation will be related to these concepts.

Figure 2.1 presents the interrelation between different modelling and simulation related concepts as introduced by Zeigler.

Object is some entity in the Real World. Such an object can exhibit widely varying behaviour depending on the context in which it is studied, the aspects of its behaviour which are under study, ...

Base Model is a hypothetical, formalised (abstract) representation of the object's properties, in particular, its behaviour, which is valid in all possible contexts, describes all the object's facets, ...
A base model is hypothetical as we will never —in practice— be able to construct/represent such a "total" model. The question whether a base model exists in theory is a philosophical one (*cfr.* hidden parameter problem in quantum physics).

System is a well defined object in the Real World under specific conditions, only considering specific aspects of its behaviour.

Experimental Frame (context) When one studies a system in the real world, the experimental frame (EF) describes experimental conditions (context), aspects, ... within which that system (and corresponding models) will be used. A description of the Experimental Frame as used in WEST++ is given below.

Lumped Model gives an accurate description of a system within the context of a given Experimental Frame. The term "accurate description" needs to be defined precisely. Usually, certain properties of the system's structure and/or behaviour must be reflected by the model (within a certain range of accuracy). Note: a lumped model is not necessarily a lumped parameter model (due to the diverse applications of modelling and simulation, terminology overlap is very common). The current state-of-the art of modelling represents models in different formalisms. Formalism transformation makes an overall system model suitable for simulation. In WEST++, the ODE (Ordinary Differential Equation), DAE (Differential Algebraic Equation), and PDE (Partial Differential Equation) formalisms are supported, enhanced with Waste Water Treatment Plant knowledge.

Experiment(ation) is the physical act of carrying out an experiment. An experiment may interfere with system operation (steer input and parameters) or it may not. As such, the experimentation environment

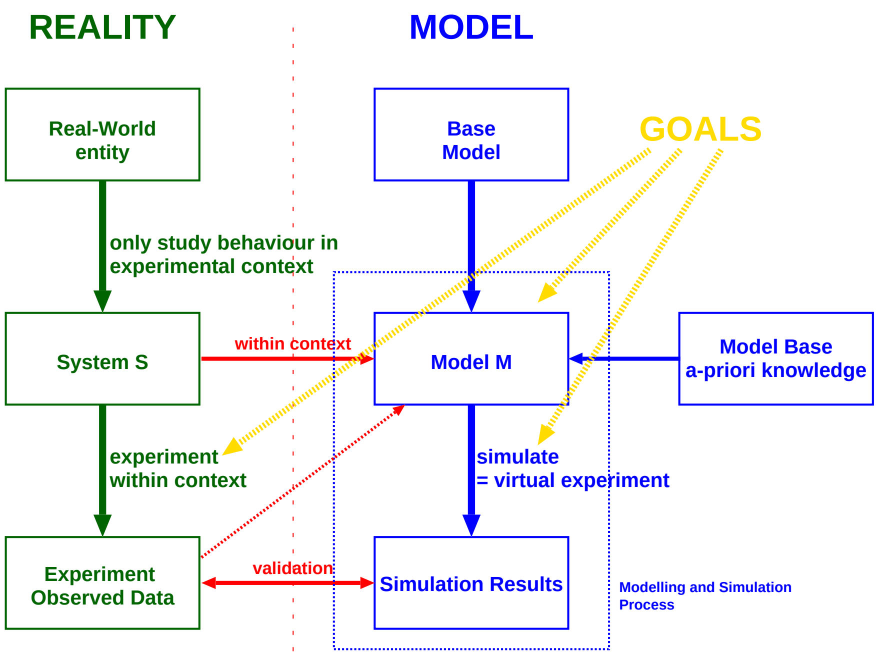


Figure 2.1: Modelling and Simulation

may be seen as a "system" in its own right (which can be modelled in a lumped model). Also, experimentation involves observation. Observation yields *Observed Data*.

Simulation of a lumped model in a certain formalism (*e.g.,* algebraic, Petri net, ...) will calculate the dynamic input/output behaviour through symbolic or numerical operations. Simulation yields *simulation results*. Note the above statement that simulation may use symbolic as well as numerical techniques. Simulation can be seen as *virtual experimentation*, and as such, the particular technique used does not matter. Whereas the goal of modelling is to meaningfully describe a system presenting information in an understandable, re-usable way, the aim of simulation is to be *fast and accurate*. Symbolic techniques are often favoured over numerical ones as they allow the generation of classes of solutions rather than just a single one (*e.g.,* $sin(x)$ as a solution to the harmonic equation as opposed to one single approximate trajectory solution). Furthermore, symbolic optimisations have a much larger impact than numerical ones thanks to their global nature.

Verification is the process of checking the consistency of a simulation program with respect to the lumped model it is derived from.

Validation is the process of comparing experiment *measurements* with *simulation results* within the context of a certain Experimental Frame. When comparison shows differences, the formal model built may not correspond to the real system. A large number of matching *measurements* and *simulation results* does *not prove* correctness of the model however. For this reason Popper has introduced the concept of *falsification*: the enterprise of trying to falsify (*i.e.,* disprove) a model.

The use of simulation has proven to be invaluable in the study of complex systems. The simulation activity is part of the larger model-based systems analysis enterprise. A framework for these activities is depicted in Figure 2.2.

The Framework starts by identifying an Experimental Frame. As mentioned above, the frame represents the experimental conditions under which the modeller wants to investigate the system. As such, it reflects the modeller's goals and questions.

Based on a frame, a class of matching models can be identified. Through structure characterization, the appropriate model structure is selected based on a priori knowledge and measurement data. Subsequently,

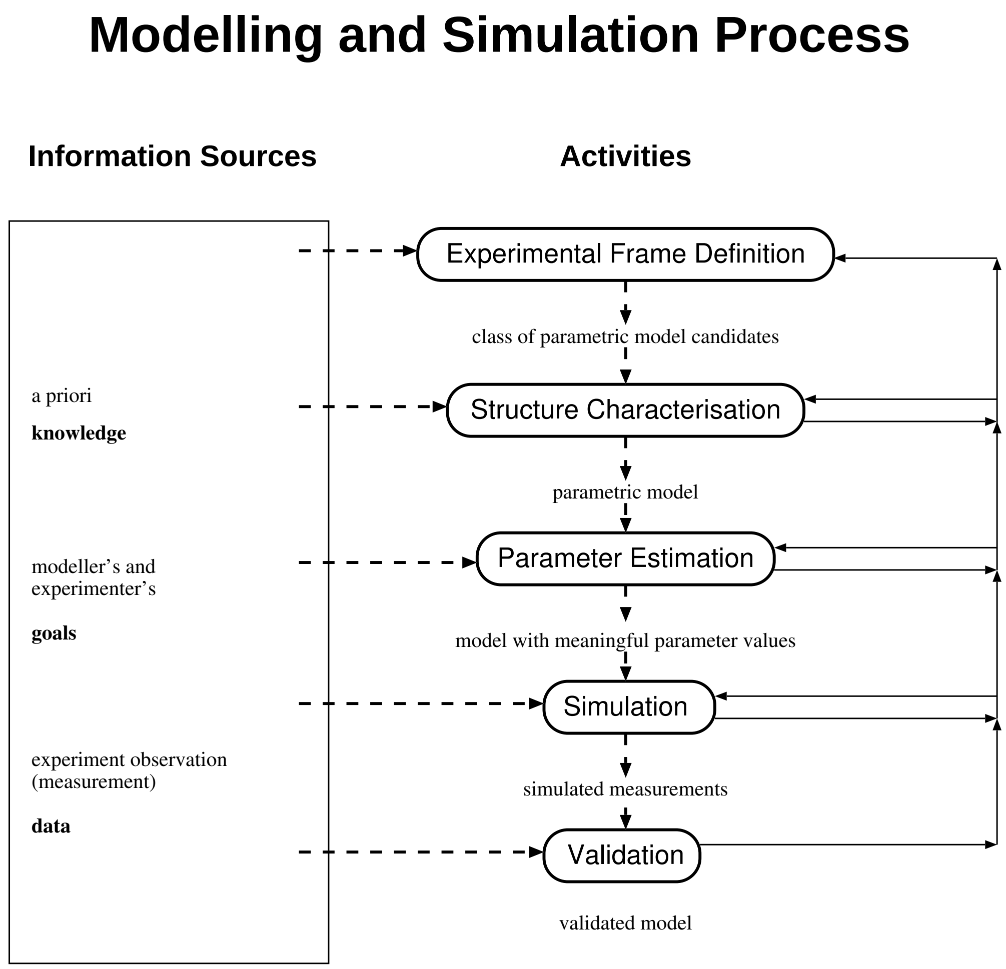


Figure 2.2: Model-based Systems Analysis

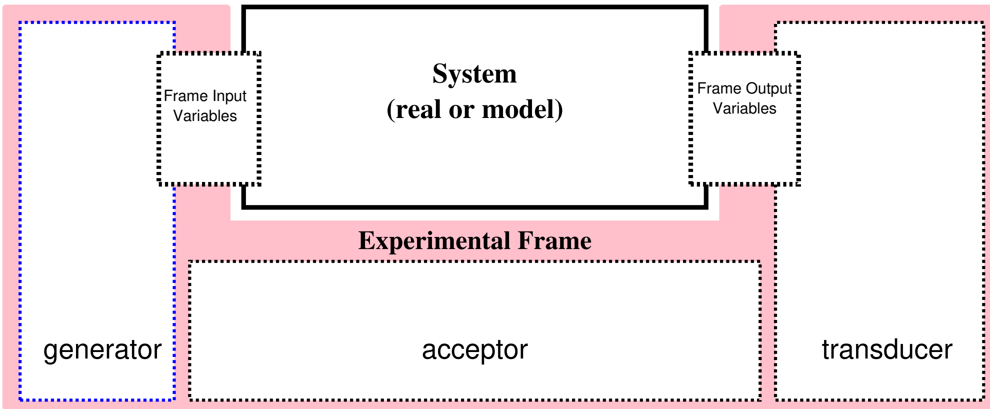


Figure 2.3: System versus Experimental Frame

parameter identification (or estimation) yields optimal parameter values for reproducing a set of measurement data. Using the identified model and parameters, simulation allows one to mimic the system behavior (virtual experimentation). It is noted that simulation can be embedded in a plethora of virtual experiments: initial value problems, shooting problems, optimisation problems, parameter fit problems, ...

The question remains however whether the model has predictive validity: is it capable not only of reproducing data which was used to choose the model and to identify parameters but also of predicting new behavior ?

In Figure 2.2, one notices how the different steps in the modelling process may each introduce errors. As indicated by the feedback arrows in Figure 2.2, a model has to be corrected once proven invalid. A very desirable feature of the validation process is the ability to provide hints as to the location of modelling errors. Unfortunately however, very few methods are designed to systematically provide such information.

The concept of Experimental Frame (see Figure 2.3) refers to a limited set of circumstances under which a system (real of model) is to be observed or subjected to experimentation. As such, the Experimental Frame reflects the *objectives* of the experimenter who performs experiments on a real system or, through simulation, on a model. In its most basic form, an Experimental Frame consists of two sets of variables, the Frame Input Variables and the Frame Output Variables, which match the system or model terminals. On the input variable side, a *generator* describes the inputs or stimuli applied to the system or model during an experiment. A generator may for example specify a unit step stimulus. On the output variable side, a *transducer* describes the transformations to be applied to the system (experiment) or model (simulation) outputs for meaningful interpretation. A transducer may for example specify the calculation of the extremal values of some of the output variables. In the above, *output* refers to physical system output as well as to the synthetic outputs in the form of internal model states measured by an observer. In particular, in case of a model, outputs may *observe* internal information such as state variables or parameters.

Apart from input/output variables, a generator and a transducer, an Experimental Frame, may also comprise an *acceptor* which compares features of the generator inputs with features of the transduced output, and determines whether the the system (real or model) "fits" this Experimental Frame (and hence, the experimenter's objectives).

The above presentation of an experimental frame enables a rigorous definition of model *validity*. Let us first postulate the existence of a unique *Base Model*. This model is assumed to accurately represent the behavior of the Real System under *all* possible experimental conditions. This model is *universally valid* as the data $D_{RealSystem}$ obtainable from the Real System is always equal to the data $D_{BaseModel}$ obtainable from the model.

$$D_{BaseModel} \equiv D_{RealSystem}$$

A Base Model is distinguished from a *Lumped Model* by the limited experimental context within which the last accurately represents Real System behavior.

A particular experimental frame E may be applicable to a real system or to a model. In the first case, the data potentially obtainable within the context of E are denoted by $D_{RealSystem}|E$. In the second case, obtainable data are denote by $D_{model}|E$. With this notation, a model is valid for a real system within Experimental Frame E if

$$D_{LumpedModel}|E \equiv D_{RealSystem}|E$$

The data equality $\equiv$ must be interpreted as "equal to a certain degree of accuracy".
The above shows how the concept of validity is *not absolute*, but is related to the experimental *context* within which Model and Real System *behavior* are compared and to the *accuracy metric* used.
One typically distinguishes between the following types of model validity:

Replicative Validity concerns the ability of the Lumped Model to *replicate* the input/output data of the Real System. With the definition of a Base Model, a Lumped Model is replicatively valid in Experimental Frame E for a Real System if

$$D_{LumpedModel}|E \equiv D_{BaseModel}|E$$

Predictive Validity concerns the ability to identify the *state* a model should be set into to allow *prediction* of the response of the Real System to *any* (not only the ones used to identify the model) input segment. A Lumped Model is predictively valid in Experimental Frame E for a Real System if it is replicatively valid and

$$F_{LumpedModel}|E \subseteq F_{BaseModel}|E$$

where F_S is the set of I/O functions of system S within Experimental Frame E. An I/O function identifies a *functional relationship* between Input and Output, as opposed to a general non-functional relation in the case of replicative validity.

Structural Validity concerns the *structural relationship* between the Real System and the Lumped Model. A Lumped Model is structurally valid in Experimental Frame E for a Real System if it is predictively valid and there exists a morphism $\stackrel{\triangle}{=}$ from Base Model to Lumped Model within frame E.

$$LumpedModel|E \stackrel{\triangle}{=} BaseModel|E$$

When trying to assess model validity, one must bear in mind that one only observes, at any time t, $D^t_{RealSystem}$, a subset of the potentially observable data $D_{RealSystem}$. This obviously does not simplify the model validation enterprise.
Whereas assessing model validity is intrinsically impossible, the *verification* of a *model implementation* can be done rigorously. A *simulator* implements a lumped model and is thus a source of obtainable data $D_{Simulator}$. If it is possible to prove (often by design) a structural realtionship (morphism) between Lumped model and Simulator, the following will hold unconditionally

$$D_{Simulator} \equiv D_{LumpedModel}$$

As mentioned before, different modelling errors may be introduced during different steps of the modelling process as depicted in Figure 2.2:

Experimental Frame Error: In defining the boundaries of the process or system to be modeled, some important components may be missed, some significant disturbances to the system may be neglected and so on. All of these introduce errors into the model.

Structural Error: Due to for instance lack of knowledge of the mechanism of the process to be modeled or due to an oversimplification of the model, one may assume a wrong model structure. Typical errors include choosing an incorrect number of state variables or incorrectly assuming non-linear behavior. Structural errors may accidentally be produced through incorrect choice of parameters (usually, 0), whereby some part of the model structure vanishes, thereby altering the model structure.

Parametric Error: Either by improper or inadequate data used for parameter identification or by inadequately designed algorithms, one may use incorrect parameter values.

There are several reasons why abstract models of systems are used. First of all, an abstract model description of a system *captures knowledge* about that system. This knowledge can be stored, shared, and re-used. Furthermore, if models are represented in a standard way, the *investment* made in developing and validating models is paid off as the model will be understood by modelling and simulation environments of different vendors for a long time to come.
Secondly, an abstract model allows one to formulate and answer *questions* about the structure and behaviour of a system. Often, a model is used to obtain values for quantities which are non-observable in the real system. Also, it might not be financially, ethically or politically feasible to perform a real experiment (as opposed to a simulation or virtual experiment). Answering of *structure related* questions is usually done by means of symbolic analysis of the model. One might for example wish to know whether an electrical circuit contains a loop. Answering of questions about the *dynamic behaviour* of the system are done (by definition) through *simulation*. Simulation may be symbolic or numerical. Whereas the aim of modelling is to provide insight and to allow for re-use of knowledge, the aims of simulation are accuracy and execution speed (often real-time, with hardware-in-the-loop).
One possible way to construct systems models (particularly in systems design) is by copying the structure of the system. Such is not a strict requirement. A neural network which simulates the behaviour of an aeration tank in an activated sludge waste water treatment plant is considered a "model" of the tank. It may accurately replicate the behaviour of the tank, though the physical structure of the tank and its contents is no longer apparent. For purposes of control, we are often satisfied with a performant (real-time) model of a system which accurately predicts its behaviour under specific circumstances, but bears no structural resemblance with the real system.

2.2 General Systems Theory

A general mathematical framework exists for the description of *causal*, *deterministic*, *state based* system models. The framework is causal as it assumes the inputs to the system are known and the system model allows, in principle, the calculation of the state and output trajectories (*i.e.,* evolution in function of time) in a *unique, deterministic fashion*.

$$SYS =< T, X, \omega, Q, \delta, Y, \lambda >$$

T	time base
X	input set
$\omega : T \to X$	input segment
Q	state set
$\delta : T \times \Omega \times Q \to Q$	transition function
Y	output set
$\lambda : Q \to Y$	output function

In this system theoretical framework, the *state* Q and *state transition function* δ need to satisfy

$$\forall t_x \in [t_i, t_f] : \delta(t_i, \omega_{[t_i,t_f]}, q_i) = \delta(t_x, \omega_{[t_x,t_f]}, \delta(t_i, \omega_{[t_i,t_x]}, q_i))$$

As depicted in Figure 2.4, this means the (state) transition from a time t_i to a time t_f can be split up into any number of intermediate state transtions and still give the same result at time t_f. This is a fundamental property which forms the basis for the implementation of *all* (*i.e.,* both continuous and discrete) simulators. For *any* deterministic, state-based model, a *solver* or *simulation kernel* will generate state and output trajectories for a given input segment. At each point in time, λ is used to calculate output from state. For each time segment, δ computes the state at the end of the time segment from the state at the initial time and from the input segment over the time segment. This is depicted in Figure 2.5 It is important to note that this

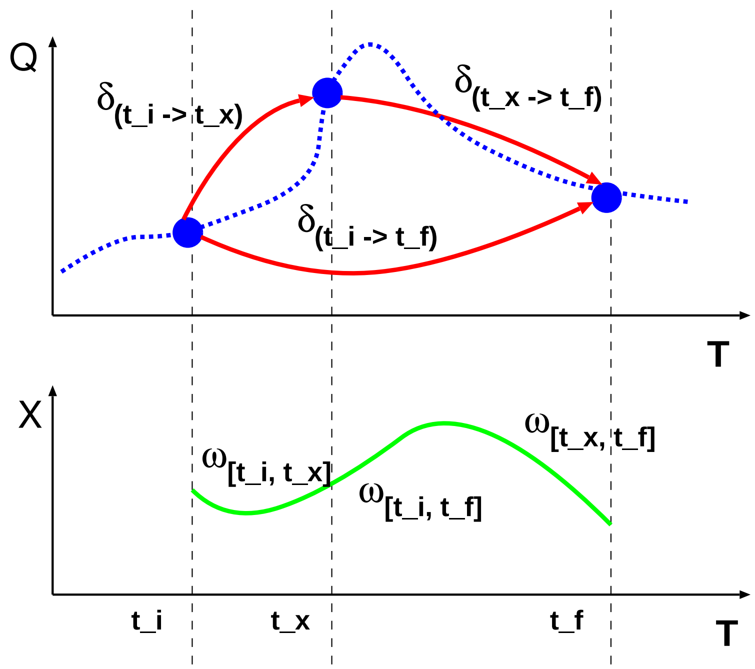


Figure 2.4: State Transition Property

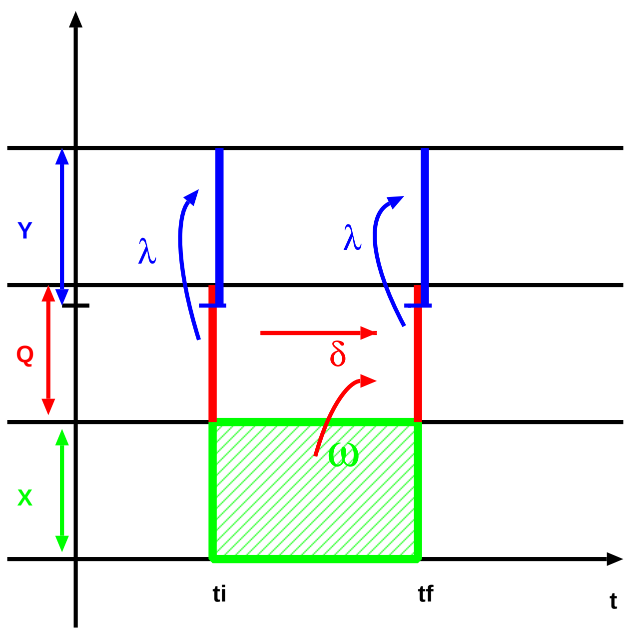


Figure 2.5: General, Deterministic, State Based Simulation Kernel

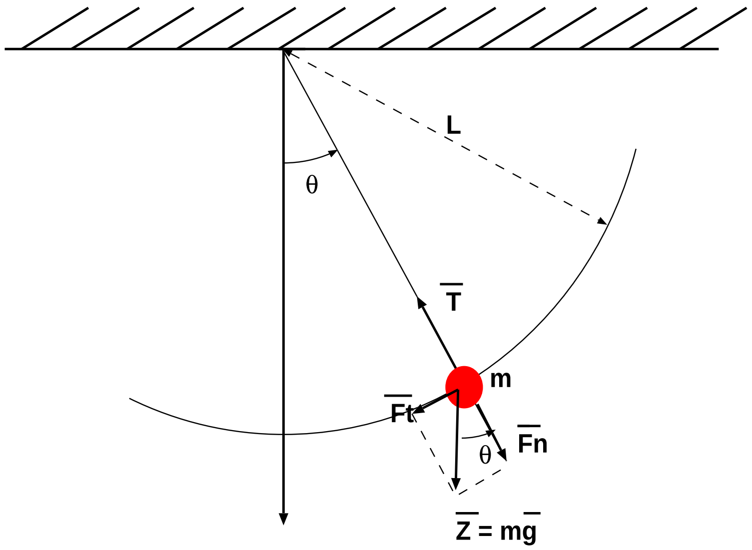


Figure 2.6: Pendulum

computation structure is the generic template for any causal, state-based simulator.
The $SYS =< T, X, \omega, Q, \delta, Y, \lambda >$ formalism captures many different specific formalisms. As a simple continuous ($T = \mathcal{R}$) example, a pendulum as depicted in Figure 2.6 fits the SYS template:

$$PENDULUM =< T, X, \omega, Q, \delta, Y, \lambda >$$

$$
\begin{array}{l}
T = \mathcal{R} \\
X = \emptyset \\
\omega : T \to X \\
Q = [-\frac{\pi}{2}, \frac{\pi}{2}] \times \mathcal{R} \\
\delta : T \times \Omega \times Q \to Q \\
\delta(t_i, \omega_{[t_i,t_f]}, (\theta(t_i), \phi(t_i))) = (\theta(t_i) + \int_{t_i}^{t_f} \phi(\alpha) d\alpha, \phi(t_i) + \int_{t_i}^{t_f} \frac{g}{L}\theta(\alpha) d\alpha) \\
Y = \mathcal{R}^+ \times \mathcal{R} \\
\lambda : Q \to Y \\
\lambda(\theta, \phi) = (L\cos\theta, L\sin\theta)
\end{array}
$$

In fact, the SYS template allows for the classification of different formalisms used to model the dynamic behaviour of systems. Classification is done based on the nature of the time set and of the state set. Some typical formalisms are depicted in Figure 2.7

2.3 A-Causal Modelling

Physical systems can often by used in different "causal contexts". For example, an electrical resistor can be used in a context where the voltage drop over it is known and one wishes to determine the current flowing through it, or the current flowing through the resistor may be known and one wishes to determine the voltage drop. It is obvious that there is only *one* physical system: the resistor, though there are multiple causal (given input, calculate output) models to describe the input/output behaviour. From a re-use point of view this is hardly elegant or efficient. Hence, it is appropriate to represent models of physical behaviour in an a-causal or implicit form (as opposed to a causal or explicit form). Below, a small demonstration is given of symbolic transformations of implicit models to an explicit form which can be simulated efficiently and accurately.

1. A general set of implicit equations is transformed to a causal form: for each –possibly non-linear– equation, we determine which variable in that equation can be uniquely determined from the other

Time Base / State	Continuous	Discrete	Empty
Continuous	DAE PDE	Difference Equations	Algebraic Eqn
Discrete	Discrete Event Naive Physics	Finite State Automata	Integer Eqn

Figure 2.7: Classification of Causal Formalisms

variables in that equation. Thus, each equation is assigned one "output" variable, whereas the remaining ones are "input" variables. The set of equations below

$$\begin{cases} x + y + z &= 0 \quad \text{Equation 1} \\ x + 3z + u^2 &= 0 \quad \text{Equation 2} \\ z - u - 16 &= 0 \quad \text{Equation 4} \\ u - 5 &= 0 \quad \text{Equation 4} \end{cases}$$

is transformed by means of "bipartite graph maximal cardinality matching" (the graph is bipartite as the equations and variables are disjoint sets) into

$$\begin{cases} x + \underline{y} + z &= 0 \quad \text{Equation 1} \\ \underline{x} + 3z + u^2 &= 0 \quad \text{Equation 2} \\ \underline{z} - u - 16 &= 0 \quad \text{Equation 4} \\ \underline{u} - 5 &= 0 \quad \text{Equation 4} \end{cases}$$

Hereby, the underlined variables need to be determined from the remaining ones. The causality assignment is not always possible. In case of failure of the causality assignment, this is an indication of a modelling error of the set has to be solved implicitly by means of a numerical solver (*e.g.,* DASSSL DAE solver).

2. The above can be transformed into explicit form through symbolic manipulation (underlined variables are underlined and brought to the left hand side).

$$\begin{cases} \underline{y} &:= -x - y \\ \underline{x} &:= -3z - u^2 \\ \underline{z} &:= u + 16 \\ \underline{u} &:= 5 \end{cases}$$

3. This still results in a *set* of equations which, when written in an arbitrary order in a programming language with sequential semantics such as C++, will lead to erronous results. Hence, the "sorting" of equations based on Tarjan's Topological Sort algorithm. This leads to

$$\begin{aligned} u &:= 5; \\ z &:= u + 16; \\ x &:= -3z - u^2; \\ y &:= -x - y; \end{aligned}$$

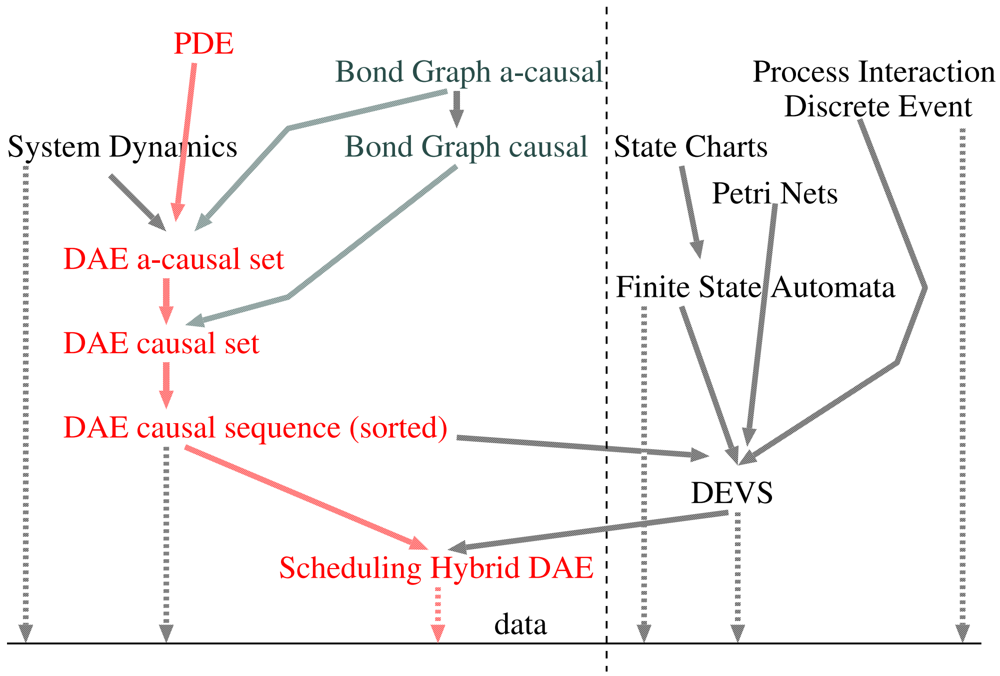


Figure 2.8: Formalism Transformation Graph

> Note how cyclic dependency between variables may occur. These "algebraic loops" need to be detected for solution by either an implicit numerical solver or –where possible– by means of symbolic manipulation (usually only in case of linear systems). The reason to not use an implicit numerical solver from the start is that such is highly inefficient (due to the iterative nature of such a solver).

The above is a small selection of about 10 symbolic transformations which are applied to DAE models in WEST++.

2.4 The Formalism Transformation Graph (FTG)

When modelling complex systems, often different abstractions are used. More formally, those abstractions are "formalisms" such as Bond Graphs, DAEs, ODEs, Finite State Automata, . . .

The Formalism Transformation Graph illustrated in Figure 2.8 represent possible "formalism transformations". The arrows denote the possibility to transform from one formalism to a "lower" one. This often entails loss of information. It should be noted that the "total" information content of the model *and* the solver usually stays constant. In the figure, the "data" level denotes the results of simulations. The dotted lines denote numerical simulation as a means of transfomration. The solid lines denote symbolic transformations. The figure charts the currently used –albeit often in hard-coded form– formalism transformations.

An algorithm was developed which determines, based on the information in the FTG, which transformations need to be applied to a complex coupled model (consisting of diverse coupled sub-models, often described in different formalisms) and in which order to "optimally" arrive at 1 simulatable model (at MSL-EXEC level, see below).

2.5 MSL-USER vs. MSL-EXEC

The aim of modelling –the representation, storage and meaningful re-use of knowledge about the dynamic behaviour of systems– on the one hand and the aim of simulation –the efficient and accurate solving of models to make system behaviour explicit in the form of state trajectories– are almost contradictary. Hence, the representation of model knowledge at a high level (in MSL-USER) on the one hand and at a low level (in MSL-EXEC) on the other hand.

2.6 MSL-USER

MSL-USER is an object-oriented language which allows for the declarative representation of system dynamics. Declarative means that the model ("what") is presented without specifying "how" to solve the model. The latter is called an operational specification. This declarative specification is automatically checked (syntax and semantics) and transformed into a low-level representation (MSL-EXEC, with C++ binding in this case) suitable for effecient (numerical) simulation. The translation involves symbolic manipulation (computer algebra) and graph manipulation.
The main characteristics of MSL-USER are:

- MSL-USER is built around a number of generic "types" corresponding to abstract mathematical concepts such as Integer number, Real number, product set, mapping, matrix, ... The "meaning" of objects of these types is given in a "denotational" fashion by referring to the corresponding mathematical concepts. By means of these types and with extra semantic rules, it becomes possible to represent different formalisms such as Petri Nets, Bond Graphs, System Dynamics, Differential Algebraic Equations (DAE), ...

- MSL-USER allows one to represent abstract models of the behaviour of (physical) systems. In particular, it is possible to represent "systems" in the system theoretical (state, transition function, output function, ...) sense.

- MSL-USER allows one to express "physical" knowledge such as units (m, kg, ...), quantitytype (Length, Mass, ...), physical nature (across, through), boundary conditions, ... The semantics of these are known to the MSL-USER compiler which will check model consistency and, where appropriate, apply this knowledge in the translation towards MSL-EXEC.

- MSL-USER allows for the declarative, non-causal (implicit) representation of models. Through causality assignment, causal (explicit) equations are obtained.

- Re-use of models becomes possible thanks to the EXTENDS inheritance mechanism. This mechanism allows for the extension of an exisiting model. Thus, starting from generic models, a tree of extended models can be built.

- Classification is made possible through the SPECIALISES mechanism. Hereby, it is possible to indicate that a particular type is a sub-type of another type. This not only allows for classification, but also for rigorous type-checking.

The MSL-USER compiler is written in lex(flex), yacc(bison), and C++ and makes intense use of LEDA (Library of Efficient Data structures and Algorithms).
Examples of MSL-USER will be given further on in this report.

2.7 MSL-EXEC

MSL-EXEC is generated automatically from MSL-USER and contains both code to describe dynamics as code to represent the symbolic model information. Figure 2.9 depicts the general structure of a "simulator". The solver communicates efficiently (vectors are being passed) with the model dynamics part of the MSL-EXEC model. In essence, the model dynamics encodes the state transition function as right hand sides of ODEs and as algebraic equations. The simulator as a whole can be asked to perform a numerical simulation. This entails using the solver to generate a state trajectory for the particular MSL-EXEC model. The simulator can also be queried for symbolic information. This will be retrieved from the symbolic information part of the MSL-EXEC model.
Below is a excerpt of an automatically generated MSL-EXEC model. In the constructor of the C++ class `MassSpringClass`, all symbolic information is registered.

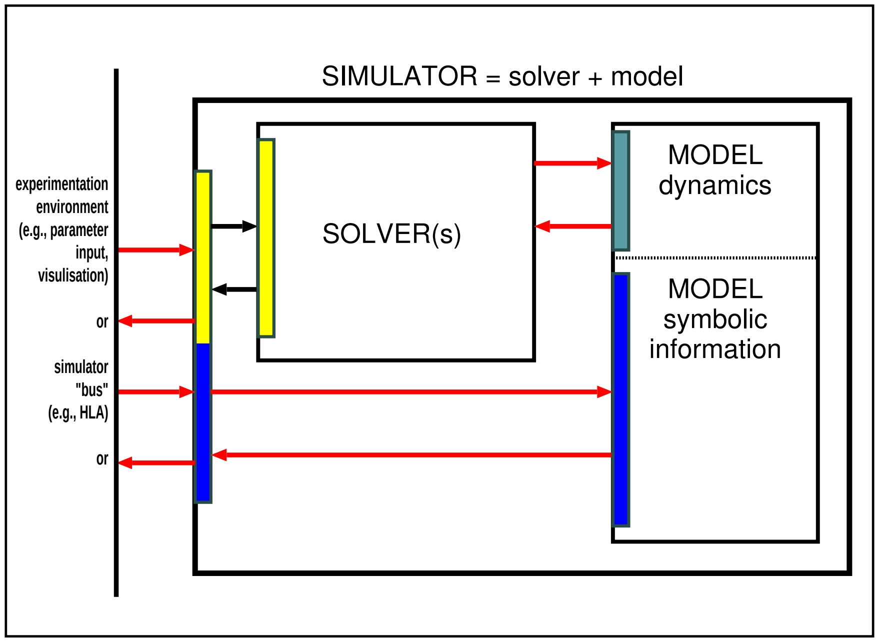


Figure 2.9: Model-Solver Architecture

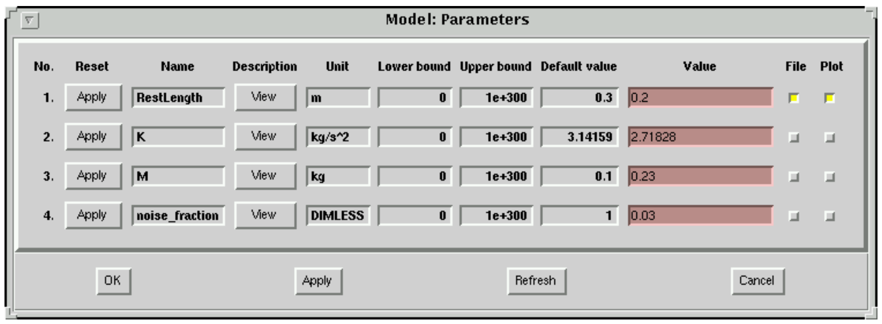


Figure 2.10: Continuous Simulation Experiment

```
MassSpringClass :: MassSpringClass(StringType name_arg)
{
  set_name(name_arg);
  set_description("Simple Frictionless Mass Spring System");
  set_class_name("MassSpringClass");

  set_no_params(4);
  set_param(0, new MSLEParamClass("RestLength", "m", 0.3, 0, MSLE_PLUS_INF,
                                  "Length of the spring \n in rest"));
  ...
}
```

This symbolic information is displayed in the WEST++ GUI as depicted in Figure 2.10.
Information –the right hand sides of equations– for the numerical solver is given in the `ComputeInitial()`, `ComputeState()`, `ComputeTerminal()`, and `ComputeOutput()` methods:

```
void MassSpringClass :: ComputeInitial(void)
{
  _factor_ = _K_/_M_;
```

```
}

void MassSpringClass :: ComputeState(void)
{
  _deviation_ = _x_ - _RestLength_;

  _D_x_ = _v_;
  _D_v_ = - _factor_ * _deviation_;
}

void MassSpringClass :: ComputeOutput(void)
{
  int random_sign;
  int max_num = 256;
  double noise;

  _x_out_ = _x_;

  random_sign = (random()&1)*2 -1;
  noise = random_sign*((random()&max_num)/max_num);
  _x_measured_out_ = _x_+_noise_fraction_*noise;
}
```

2.8 Solvers

As depicted in Figure 2.11, a general solver integrates the state equations, possibly adapting the integrator stepsize. Events are handled by a discrete-event kernel.

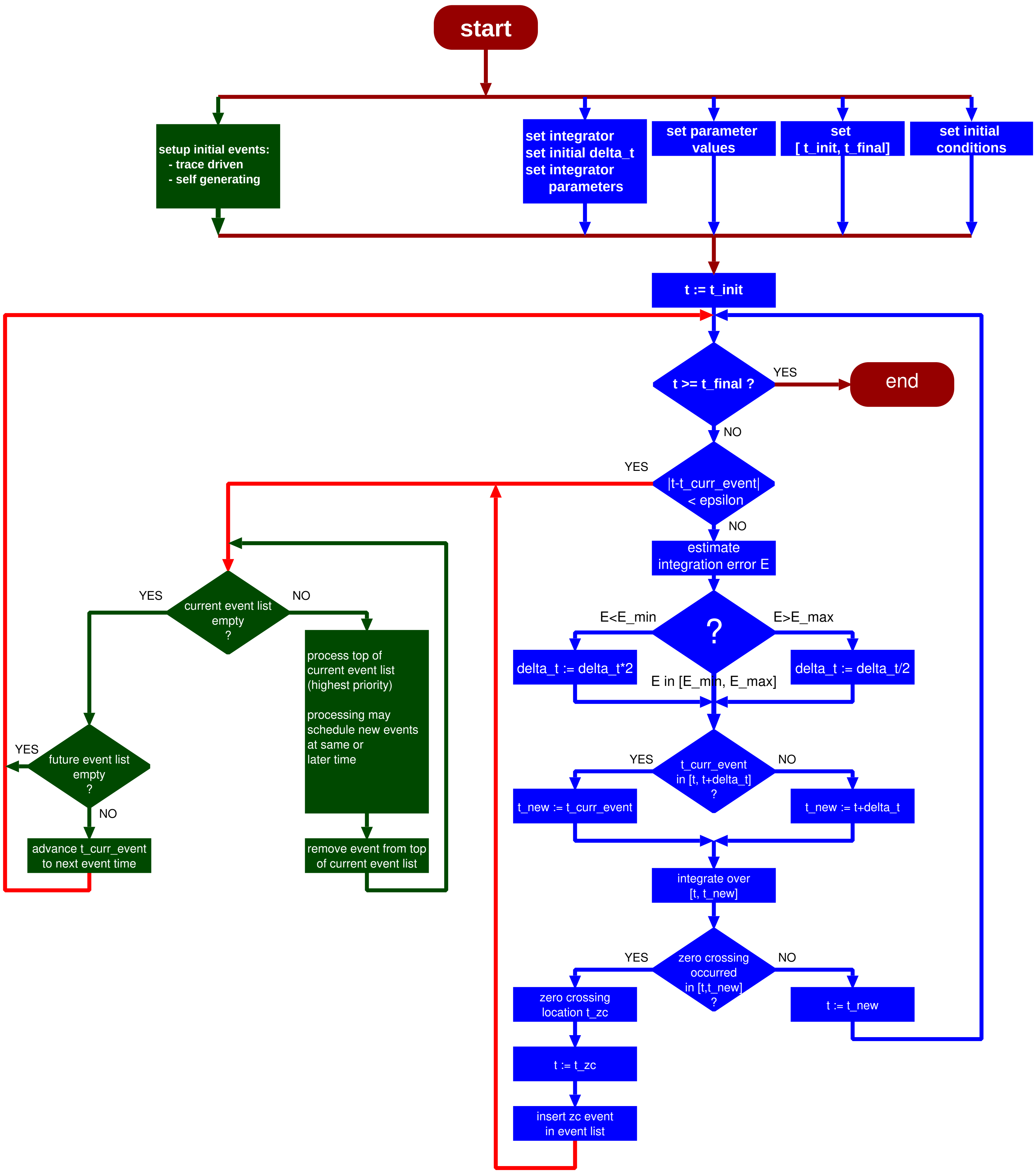


Figure 2.11: Hybrid Simulation Kernel

3

The WEST++ Modelling and Simulation Environment

The WEST++ Modelling and Simulation Environment implements the concepts presented in the previous chapter. The architecture makes a strict distinction between a *modelling environment* which aims to provide insight and enable re-use of model knowledge, on the one hand and the experimentation environment which aims to maximise accuracy and performance.

3.1 Modelling

As depicted in Figure 3.1, the WEST++ modelling environment allows for component based modelling (in this case of a simple Waste Water Treatment Plant): the user connects model icons in a hierarchical fashion. From this abstract specification, together with an MSL-USER library of dynamic models, one single MSL-USER model is produced. A compiler then generates MSL-EXEC from this model for use within the experimentation environment.

3.2 Experimentation

The following distinguishes, based on general modelling and simulation theory, between different ”experiment types” as implemented within the WEST++ environment. It is important to make a strict distinction between the user view of an experiment (experiment types) and the kernel view of an experiment (the solvers used to implement the experiments). It is argued that all but expert users should be presented a ”user view” of the system rather than a ”kernel view” in the interactive Graphical User Interface.

3.2.1 Kernel View

The kernel consists of a number of ”solvers” which, when linked to a ”model” (in MSL-EXEC), and given some parameters pertaining to the kernel's operation (such as accuracy, termination conditions, ...), will ”solve” that model. The ”model” is just a ”standard” defined representation (in C) of the mathematical object (such as an ODE) to be solved. In WEST++, a general MSL-EXEC model may contain an arbitrary sequence of Explicit Algebraic and Ordinary Differential Equations (ALGODE), (IALG) Implicit Algebraic Equations, and Implicit Differential Algebraic Equations (DAE).
In the WEST++ context, the following kernels are used:

1. Explicit ALGODE solver

   This solver integrates a sorted sequence of explicit algebraic and ordinary differential equations.

$$\frac{dx}{dt} = f(x', x, y, u, p, t) y = g(x', x, y, u, p, t)$$

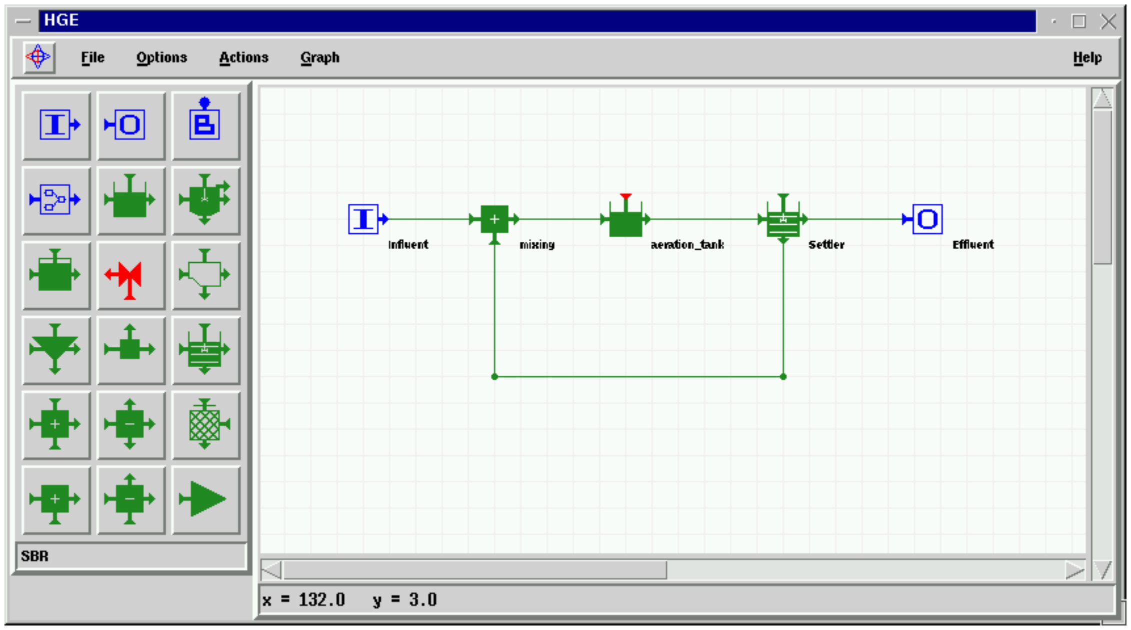


Figure 3.1: Simple WWTP Model

wereby x is a vector of derived state variables y is a vector of algebraic variables u is a vector of known inputs p is a vector of parameters (constant during one simulation run) t is the independent variable. Integration is done over $[t_{in}, t_{fin}]$ f is a, possibly non-linear, vector function (image dimension is equal to the dimension of x) g is a, possibly non-linear, vector function (image dimension is equal to the dimension of y)

The above mathematical representation is extended with the requirement that all individual (scalar) equations are sorted: for each equation, the right hand side only depends on variables which are available from (left hand sides of) preceding equations.

To solve the model, initial conditions (for the state variables) as well as t_in and t_fin must be given.

The integrator's stepsize may be variable to increase computation speed. Changes in stepsize are driven by an accuracy estimate computed by the solver. Such an accuracy estimate is usually based on an estimate for the stepwise error which is computed from the difference between two subsequent order approximations.

2. Implicit ALG solver

This solver takes an implicit algebraic model

$$0 = g(x', x, y, u, p, t)$$

whereby the x and x' are assumed known (as well as p, u, and t) and tries to find a y^* for which the above equation holds. The above equation is represented in MSL-EXEC in the form of residuals

$$r = g(x', x, y, u, p, t)$$

The solver tries to make the residuals 0 (in a manner which is not unlike what an optimiser does).

In WEST++, different solvers are used for an implicit algebraic model of dimension 1 and one of higher dimension.

3. Implicit DAE solver

This solver takes

$$0 = g(x', x, y, u, p, t)$$

and solves it for x and y (u, p, and t are known).

In WEST++, Linda Petzold's DASSL solver is used.

4. Optimiser

This solver takes a real-valued goal function

$$G(v)$$

and tries to vary the v as to extremise (minimise or maximise) the goal. Possibly, optimisation can be constrained by algebraic inequalities.

Note how an optimiser tries to extremise a single real value. If multiple criteria need to be optimised, an appropriate weighting of these needs to be encoded in the goal function.

As the result of a simulation is usually a trajectory (x(t), y(t)) over $[t_in, t_fin]$, the goal function can make use of all x and y values in that interval. Typical uses are:

- use only values at t_fin (shooting problem)
- use the integral over time of a trajectory (total cost)
- use a discrete sum of values at particular time instants (comparing simulated values with measured values at those time instants).

WEST++ uses Brent's Praxis as well as a Simplex method.

5. Miscellaneous Matrix Solvers

Eigenvalues, eigenvectors, matrix inversion, ... are often used (*e.g.,* in control design or in sensitivity analysis).

In WEST++, standard matrix solvers are used as "external function" calls.

Certain solvers are able to increase performance and/or accuracy by making use of "extra" information (such as the Jacobian of the model) provided in the MSL-EXEC model. Such information may have been arrived at through symbolic manipulation of the original model.
In the above we have not yet mentioned the handling of "events". An "event" interrupts the "continuous" integration due to some condition (see below). The event handling then takes care of the appropriate event code. The event happens "instantaneously": simulated time does not advance. An event will typically

- modify state variables (continuous as well as discrete "mode" variables). Typically, a continuous state variable will be given a new value wherafter the continuous integration resumes (a "re-start") from a consistent set of initial values.
- "schedule" an event to take place at some future time
- "schedule" an event at the same time: as an abstraction of real-world process, a chain of instantaneous events may have to be iterated through. A typical example of this is when the behaviour of a discrete controller is modelled using a Finite State Automaton.

Events at a single event time are handled in a highest priority first, First In First Out (FIFO) for equal priority, order.
Usually, a distinction is made between

- Time Events: the time of the event is known in advance. Typical time events are the termination of the simulation at $t_i n$ and the synchronisation with external inputs which are given at particular time instants.
- State Events: the time of the event is implicitly given by a condition such as $(v + w < 5)$ whereby v and w are state variables. The condition is called a ”zero crossing” ($v + w - 5$ crosses zero) or ”indicator” function.

3.2.2 User View

The user thinks in terms of different ”virtual experiments” on the model of a system. Typically, the sequence of these experiments, the ”modelling and simulation life-cycle” or ”modelling and simulation process” (see elsewhere for more details) comprises:

- structure characterisation
- sensitivity analysis w.r.t. parameters
- parameter estimation
- simulation (either initial value or shooting problem)
- optimisation of a goal function using the model with known parameters
- sensitivity analysis w.r.t. inputs

The following experiment types are implemented in WEST++:

1. Simulation Experiment (Initial Value and Shooting)

   Mathematically, a model is specified by

   - EQ: its equations
   - IV or TV: Initial or Terminal Values
   - BC: Boundary Conditions (in case of a system with more than one independent variable)

   Currently, in WEST++, there are two types of simulation experiments:

   - Initial Value problem: state variable values are given at t_{in}. The simulator calculates the trajectory over $[t_{in}, t_{fin}]$. This is usually implemented using a forward integrator.
   - Terminal Value, or End Value, or ”Shooting” problem: state variable values are given at t_{fin}. The simulator calculates the trajectory over $[t_{in}, t_{fin}]$. As a side-effect, the state variable values at t_{in} result. These could be used as initial values in an Initial Value problem. This is implemented in WEST++ using an optimisation algorithm whereby the varied entities are the unknown initial conditions and the goal function is the sum of absolute values of differences between simulated end-value and desired end-value.

   In theory, a mix of both is possible.

   Sometimes it is necessary to ”synchronise” with external data. This is for example the case when the input u(t) is given as a table of measurements. Two options are available:

   (a) the integrator can be forced to ”synchronise” with the external data. This is done by means of time-events (as the input times are known in advance from the table).

   (b) the integrator can determine its own integration times and when a value is needed –at those times–, interpolation is used (0-th, or 1-st order). When the input is given as a continuous function, no interpolation is required.

   Obviously, the latter solution is faster, though possibly incorrect.

2. Parameter Estimation Experiment

   Certain (determined by user, possibly after hints from model sensitivity –w.r.t. parameter variations– analysis) model parameters are varied to minimise the ”distance” between a simulated trajectory and a given (measured) trajectory. The distance measure is typically a sum of squares of differences between measured and simulated values though absolute values are also used. Differences between measured and simulated values can be calculated at different points in time: as described above, the integrator can be forced to ”synchronise” with the external data, or interpolation can be used. In general, the differences can be weighted.

3. Optimisation Experiment

   The most general use of the optimiser where some parameters are varied (possibly constrained) as to extremise a goal function.

4. Sensitivity Analysis Experiment

   The sensitivity of the model with respect to initial value or parameter variations can be investigated. If sufficient symbolic information (partial derivatives) is available (provided in the MSL-EXEC by the MSL-USER compiler), this is straightforward. If this information is not available, either a Monte Carlo technique or some local numerical technique (typically assuming local quadratic behaviour) can be used.

4
Application Area

WEST++ is a modeling and simulation environment for any kind of process which can be described (in an Object–Oriented fashion) as a structured collection of Differential Algebraic Equations (DAEs). Currently, however, WEST++ is applied to Waste-Water Treatment Plant (WWTP) design and optimization.
The PSYCO symbolic control design toolbox described in this report is demonstrated and validated in subsequent chapters by means of WWTP models as used in WEST++. In particular the IAWQ1 model (4.2) is the basis for a simple model of the dynamics of a basic WWTP.

4.1 Wastewater Treatment Plants

Environmental concerns become increasingly important in most industrialized countries. In particular, water quality is deteriorating rapidly. This fact has generated interest (both legislative and scientific) in the deployment of Waste-Water Treatment Plants (WWTPs).
The problem of modeling and simulation of wastewater treatment plants is gaining importance as a result of growing environmental awareness. Compared to the modeling of well-defined (*e.g.,* electrical, mechanical) systems, ill-defined systems modeling is more complex. In particular, choosing the "right" model is a non-trivial task.
The wastewater treatment process dealt with in WEST++ is of the "activated sludge" type. This means the reduction of waste is performed by bacteria which consider (non-toxic) waste components as food. The general WWTP structure (configuration) in its simplest form is shown in Figure 4.1. In real-life plants, many more components will be present (*cfr.* section 4.2). The basic operation principles remain the same however.
The time-varying load (influent) enters the biological reactor where biodegradable components are converted by the catalytic biomass to gaseous products. Because of the consumption of oxygen in the reactor, it is often called "aeration tank".
The effluent stream of the bioreactor is sent to the settler where both clarification and sludge thickening (through sedimentation) take place. The effluent of the settler is the clarified wastewater. This clarified effluent is typically returned to the natural environment (river, lake, . . .).
A recycle flow of sludge from the settler to the bioreactor keeps the biomass inside the system.
The (non-linear) dynamics and properties of the biological processes are still not very well understood. As a consequence, a unique model cannot always be identified. This, in contrast to traditional mechanical and electrical systems where the model can be derived from physical laws. Also, the calibration of wastewater treatment models is particularly hard: many expensive experiments may be required to accurately determine model parameters.
One way to construct system models is to copy the structure of the system. Though other approaches are

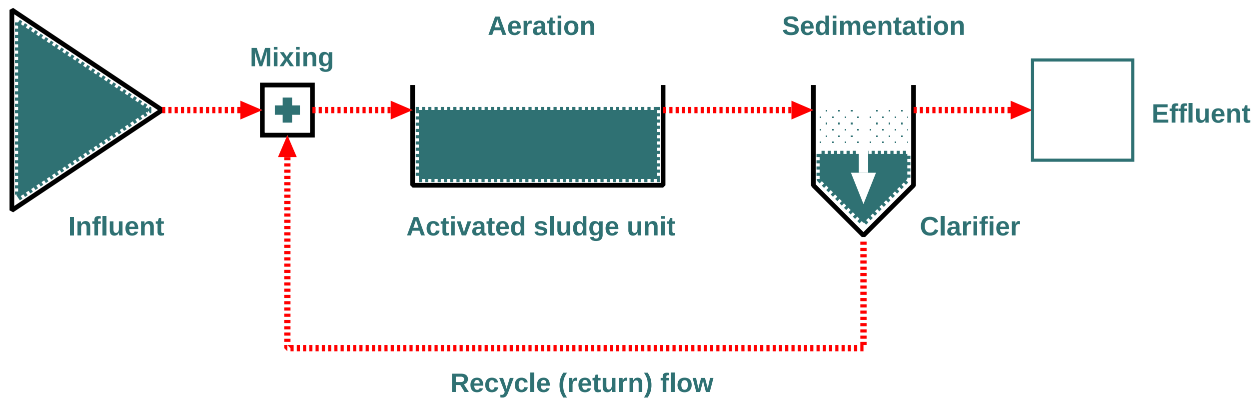


Figure 4.1: WWTP configuration

possible [11], it is common when modeling WWTPs.
Increasingly, the "system" modeled transcends the WWTP and includes the "environment" (in the engineering sense). The WWTP model is integrated in the overall model of a polluting plant, or of the river (and its natural water purification properties or toxicity tolerance) in which the effluent is dumped.
A pertinent example of explicitly modeling the environment in WWTPs concerns the "experimentation environment". Often, sensors are used to monitor biological variables. Those variables are difficult to measure, resulting in non-linear distortions of values by the sensor or by delays due to sampling and processing. Obviously, failure to include an explicit model of the sensor will completely distort results (no matter how excellent the actual system model is). This is particularly bad if one wants to estimate model parameters by fitting simulation results to measured data (the sensor behavior will be lumped into the parameters).
In general, models may be represented using different abstractions (*e.g.,* Algebraic (ALG), Ordinary Differential Equations (ODE), Petri Nets, Bond Graphs, ...). Currently, WWTP models are all of the ALG+ODE type.

4.2 Activated Sludge

The principle of activated sludge is that in a reactor a community of micro-organisms is constantly supplied with organic matter and oxygen. The micro–organisms consume the organic matter and transform it by means of aerobic metabolism, partly into new microbial biomass and partly into carbon dioxide, water and minerals. The flow of water brings about a constant wash out of the micro–organisms from the reactor to the settler. Here, the micro-organisms which grow in flocs and have acquired a density sufficient to decant, are retained and removed with the underflow. Part of this sludge is then recycled to provide biomass to treat the new influent. The surplus amount is wasted.
In many respects, the aeration basin is comparable to a conventional fermentation reactor or chemostat. However, the purpose of the process is not to produce microbial biomass or a particular metabolite, but to mineralize incoming waste materials as much as possible. It is hereby of paramount importance to minimize biomass production since the latter has to be removed and treated in a subsequent phase.
Two important characteristics further distinguish the activated sludge system from conventional microbial fermentations. First, the active biological component comprises not a pure culture but an association of bacteria, yeast, fungi, protozoa and higher organisms as rotifers. These organisms grow on the incoming waste and interact with one another. Second, the sludge consists, in contrast with its qualification "active", for an important part out of dead cells and cell debris. Indeed, young active microbial cells tend to grow in a dispersed way. The system is therefore operated in such a way that the substrate is limiting and the microbial biomass is quasi starving. Under these conditions cells grow slowly and in flocs. Because of this, the water in the decantor separates in a clear supernatant and a thick layer. Hence, the crucial part of activated sludge treatment is to select a microbial community which mineralizes at a fair rate the incoming waste and thereby produces a minimum of new biomass which furthermore sediments readily and completely out of the water

when the latter reaches the decantor.

4.3 Activated Sludge Model No.1 (IAWQ1)

Wastewater treatment practice has now progressed to the point where the removal of organic matter, nitrification and nitrogen removal by biological denitrification, can be accomplished in a single sludge system. Because of the interactions within such systems, the mathematical models describing them are quite complex, which has detracted from their use. This is unfortunate because it is with such complex systems that the engineer has the most to gain from the use of mathematical models.

Realizing the benefits to be derived from mathematical modeling, while recognizing the reluctance of many engineers to use it, the International Association on Water Pollution Research and Control (IAWPRC) formed a task group in 1983 to promote the development of, and facilitate the application of, practical models to the design and operation of biological wastewater treatment systems. The goal was first to review existing models and second to reach a consensus concerning the simplest one having the capability of realistic predictions of the performance of single sludge systems carrying out carbon oxidation, nitrification, and denitrification. The model was to be presented in a way that made clear the processes incorporated into it and the procedures for its use.

The task group chose a matrix format for the presentation of the model. The first step in setting up the matrix is to identify the components of relevance in the model. The second step in developing the matrix is to identify the biological processes occurring in the system; *i.e.,* the conversions or transformations which affect the components listed.

Within a system, the concentration of a single component may be affected by a number of different processes. An important benefit of the matrix representation is that it allows rapid and easy recognition of the fate of each component, which aids in the preparation of mass balance equations. The basic equation for a mass balance within any defined system boundary is :

$$Input - Output + Reaction = Accumulation \equiv \frac{dMass}{dt}$$

The input and output terms are transport terms and depend upon the physical characteristics of the system being modeled. The system reaction term can be easily obtained from the matrix terms.

Another benefit of the matrix is that continuity may be checked by moving across the matrix, provided consistent units have been used because then the sum of the stoichiometric coefficients must be zero.

13 components are considered in the model: soluble and particulate inert organic matter, readily and slowly biodegradable substrate, active heterotrophic and autotrophic biomass, particulate products arising from biomass decay, oxygen, nitrate and nitrite nitrogen, $NH_4^+ + NH_3$ nitrogen, soluble biodegradable organic nitrogen, particulate biodegradable organic nitrogen; the process are 8: aerobic and anoxic growth of heterotrophs, aerobic growth of autotrophos, ‘decay’ of heterotrophs and autotrophs, ammonification of soluble organic nitrogen, ‘hydrolysis’ of entrapped organics and of entrapped organic nitrogen.

Typical parameter ranges, default values, and affects of environmental factors are supplied [6].

Part II

PSYCO

5 Introduction

This part is a report about the MuPAD program PSYCO. This program gives tools to manipulate and study models from a control point of view in a symbolic (as opposed to numerical) fashion. Figure 5.1 charts the interrelationships between different code modules.
It is assumed that a state space representation or a transfer function of the model is available. The following MuPAD code presents the user with that choice:

```
// ***********************
//      Main menu
// ***********************

repeat
  print();
  print(Unquoted, "Options: ");
  print(Unquoted, "a - State variable system input");
  print(Unquoted, "b - Transfer function input");
  print(Unquoted, "q - quit");
  print();
  input("",test_main);
  case test_main
    of a do state_variable_input(): break;
    of b do transfer_function_input(): break;
  end_case;
until test_main=q end_repeat;

return();
```

The PSYCO program is composed of several routines; each routine contains more procedures and is described in a chapter of this part; each chapter has three sections:

1. **Theoretical basis** theory explanation
2. **MuPAD code** MuPAD program written to carry out the above theory
3. **Example application** MuPAD program applied to a simple system

Finally, chapter 16 contains the common procedures used by several routines.

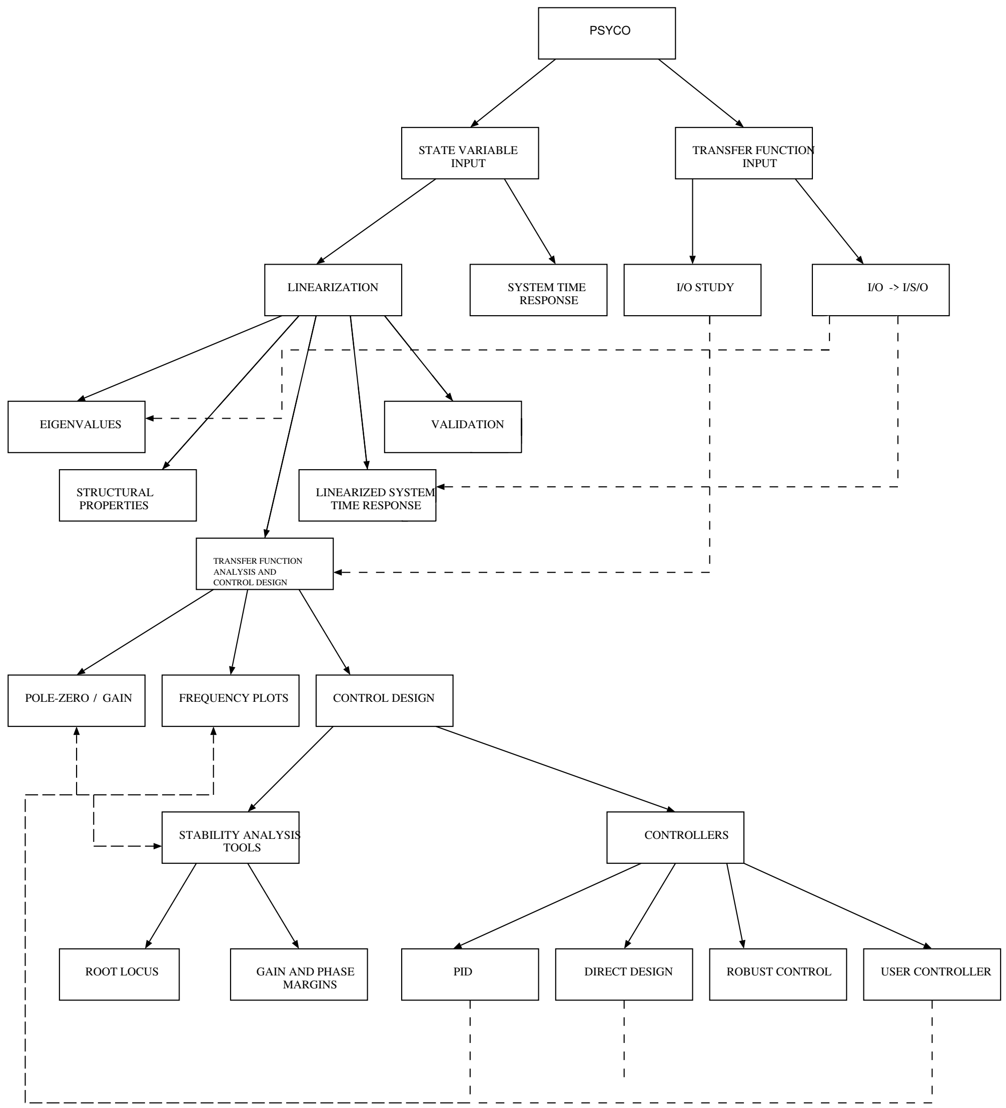


Figure 5.1: PSYCO

6

State Space Representation Input

6.1 Theoretical Basis

We assume that a state space representation of a nonlinear model is available:

$$\begin{cases} \frac{dX}{dt} = f(X,U) \\ Y = g(X,U) \end{cases}$$

where $X \in R^n$, $U \in R^m$, $Y \in R^p$ and $f : R^n \times R^m \to R^n$, $g : R^n \times R^m \to R^p$ are nonlinear functions of X and U. X are the state variables, U the (known) inputs, Y are the outputs, and t is the independent variable “time”.

6.2 MuPAD Code

The procedures used are:

- state_variable_input()
- sys_input()
- eqns_print(F,G); see section 16.5

```
sys_input:=proc()
local test;
begin

    // Nonlinear system input (MIMO systems)
    // This step allows the input of a nonlinear system Multiple
    // Input Multiple Output. The following quantities have to be
    // specified:
    // - state variables and relative equations
    // - input variables and relative nominal values
    // - output variables and relative equations
    input("State variables number: ", n):
    // X[] contains the state variable labels
    // F[] contains the right hand side of the state nonlinear system
    X:=M(n,1):
    F:=M(n,1):
```

```
    repeat
      for i from 1 to n do
        print();
        X[i,1]:=input("State variable label: ");
        print();
        F[i,1]:=input("Right hand side of the state nonlinear equation: ");
      end_for;
      print(X);
      print(F);
      input("Are these data right ? (y/n): ", test):
    until test=y end_repeat;

    print();
    input("Input variables number: ", m):
    // U[] contains the input variable labels
    // Unom[] contains the input nominal values
    U:=M(m,1):
    Unom:=M(m,1):
    repeat
      for i from 1 to m do
        print();
        U[i,1]:=input("Input variable label: ");
        print();
        Unom[i,1]:=input("Input nominal value: ");
      end_for;
      print(U);
      print(Unom);
      input("Are these data right ? (y/n): ", test):
    until test=y end_repeat;

    print();
    input("Output variables number: ", p):
    // Y[] contains the output variable labels
    // G[] contains the right hand side of the output nonlinear
    // system
    Y:=M(p,1):
    G:=M(p,1):
    repeat
      for i from 1 to p do
        print();
        Y[i,1]:=input("Output variable label: ");
        print();
        G[i,1]:=input("Right hand side of the output nonlinear equation: ");
      end_for;
      print(Y);
      print(G);
      input("Are these data right ? (y/n): ", test):
    until test=y end_repeat;

end_proc:


state_variable_input:=proc()
local test;
begin

  // State variable system input and options menu
  sys_input():
  repeat
    eqns_print(F,G):
    print();
    print(Unquoted, "Options: ");
```

```
    print(Unquoted, "a - Linearization");
    print(Unquoted, "b - System time response");
    print(Unquoted, "x - Exit");
    print();
    input("",test);
    case test
      of a do linearization(): break:
      of b do sys_time_response(): break:
    end_case:
  until test=x end_repeat:

end_proc:
```

6.3 Example Application

We consider a state space representation of a SISO (Single Input Single Output) nonlinear model, based on the Monod equation:

$$\begin{cases} \frac{ds}{dt} = -\frac{1}{y}\frac{\mu s}{k_s+s}x - sd + s_i d \\ \frac{dx}{dt} = \frac{\mu s}{k_s+s}x - bx - xd \end{cases}$$

where x, s denote the biomass and substrate concentrations in a bioreactor, respectively; s_i denotes the substrate concentration in the influent; d is the flow rate / volume ratio; y is the yield coefficient; b is the biomass decay coefficient; μ is the maximum specific growth rate, k_s is the half saturation coefficient of the substrate s.

We have U=d and we can chose Y=s (i.e. g(X,U)=s). Let us set the following,physically meaningful, parameter values:

$s_i = 10mg/l \;\; y = 0.6 \;\; b = 0.02day^{-1} \;\; \mu = 1d^{-1} \;\; k_s = 2mg/l$

This leads to:

$$\begin{cases} \frac{ds}{dt} = -\frac{1}{0.6}\frac{s}{2+s}x - sd + 10d \\ \frac{dx}{dt} = \frac{s}{2+s}x - 0.02x - xd \end{cases}$$

7
Linearization

7.1 Theoretical Basis

We have to find the equilibrium points (X=X_e such that $\frac{dX}{dt} = 0$): set $U = \overline{U}$ ($\overline{U}$: nominal value of the inputs) we have to solve a system of nonlinear equations:

$$f(X, U) = 0$$

From systems theory, the corresponding linearized system in the equilibrium point $X = X_e,\ U = \overline{U}$ is:

$$\begin{cases} \frac{d\Delta X}{dt} = \frac{\partial f(X,U)}{\partial X}|_{(X_e,\overline{U})}\Delta X + \frac{\partial f(X,U)}{\partial U}|_{(X_e,\overline{U})}\Delta U \\ Y = \frac{\partial g(X,U)}{\partial X}|_{(X_e,\overline{U})}\Delta X + \frac{\partial g(X,U)}{\partial U}|_{(X_e,\overline{U})}\Delta U \end{cases}$$

where $\Delta X = X - X_e$ and $\Delta U = U - \overline{U}$. To get a simpler notation, let us substitute ΔX with X and ΔU with U remembering that these variables are now differences from $X_e, \overline{U}$. We can rewrite such a linearized model as follows:

$$\begin{cases} \frac{dX}{dt} = AX + BU \\ Y = CX + DU \end{cases}$$

where

$$\begin{array}{ll} A = \frac{\partial f(X,U)}{\partial X}|_{(X_e,\overline{U})} & B = \frac{\partial f(X,U)}{\partial U}|_{(X_e,\overline{U})} \\ C = \frac{\partial g(X,U)}{\partial X}|_{(X_e,\overline{U})} & D = \frac{\partial g(X,U)}{\partial U}|_{(X_e,\overline{U})} \end{array}$$

The local stability of the equilibrium point can be obtained by finding the eigenvalues of A, i.e. the roots of $det[\lambda I - A] = 0$.

Structural properties (Reachability/Observabilty) can be checked by the reachability (R) and the observability (O) matrices:

$$R = \begin{bmatrix} B & AB & A^2B & \dots & A^{n-1}B \end{bmatrix}$$

$$O = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

The system is reachable (i.e. controllable) if R has full rank n.
The model is observable if O has full rank n.

7.2 MuPAD Code

The procedures used are:

- linearization()
- symbolic_linearization()
- equilibrium_point_input()
- symbolic_linearization_in_Xe()
- linearized_eqns_compute()
- eigenvalues(); see section 16.7
- reach_obs_menu()
- compute_R()
- compute_O()
- total_r_o()
- U_i_choice()
- R_i_compute()
- single_r()
- Y_i_choice()
- Ob_i_compute()
- single_o()
- single_r_o()

```
symbolic_linearization:=proc()
begin

    // Symbolic linearization.
    // This step allows to get the linearized model of a nonlinear
    // model stored in F and G. The linearized model has the form:
    //
    //             dX/dt = A*X + B*U
    //                 Y = C*X + D*U
    A:=M(n,n):
    for i from 1 to n do
      for j from 1 to n do
        A[i,j]:=diff(F[i,1], X[j,1]):
      end_for:
    end_for:
    print(Unquoted, "A = ", A);

    B:=M(n,m):
    for i from 1 to n do
      for j from 1 to m do
```

```
        B[i,j]:=diff(F[i,1], U[j,1]):
      end_for:
    end_for:
    print(Unquoted, "B = ", B);

    C:=M(p,n):
    for i from 1 to p do
      for j from 1 to n do
        C[i,j]:=diff(G[i,1], X[j,1]):
      end_for:
    end_for:
    print(Unquoted, "C = ", C);

    // WARNING: We can not use the D label as identifier because
    // it is a MuPAD keyword. We shall use Di

    Di:=M(p,m):
    for i from 1 to p do
      for j from 1 to m do
        Di[i,j]:=diff(G[i,1], U[j,1]):
      end_for:
    end_for:
    print(Unquoted, "D = ", Di);

end_proc:


equilibrium_point_input:=proc()
local test;
begin

    //  Equilibrium point input (REDUCE program should be used for
    //  this purpose). REDUCE should solve a  nonlinear system of equations
    //  with the input vector U set to its nominal value Unom

    // Xe contains the state equilibrium point
    Xe:=M(n,1):
    repeat
      for i from 1 to n do
        print();
        print(Unquoted, "Equilibrium value of ".expr2text(X[i,1]));
        Xe[i,1]:=input("");
      end_for;
      print(Xe);
      input("Are these data right ? (y/n): ", test):
    until test=y end_repeat:

end_proc:


symbolic_linearization_in_Xe:=proc()
begin

    // Linearized model evaluation in the equilibrium point

    // Aee represents A evaluated in Xe, but not in Unom yet
    Ae:=M(n,n):
    Aee:=subs(A, [  X[i,1]=Xe[i,1]    $ hold(i)=1..n ] ):
    Ae:=subs(Aee, [  U[i,1]=Unom[i,1]    $ hold(i)=1..m ] ):
    print(Unquoted, "Ae = ", Ae);

    Be:=M(n,m):
```

```
    Bee:=subs(B, [  X[i,1]=Xe[i,1]    $ hold(i)=1..n ] ):
    Be:=subs(Bee, [  U[i,1]=Unom[i,1]    $ hold(i)=1..m ] ):
    print(Unquoted, "Be = ", Be);

    Ce:=M(p,n):
    Cee:=subs(C, [  X[i,1]=Xe[i,1]    $ hold(i)=1..n ] ):
    Ce:=subs(Cee, [  U[i,1]=Unom[i,1]    $ hold(i)=1..m ] ):
    print(Unquoted, "Ce = ", Ce);

    Die:=M(p,m):
    Diee:=subs(Di, [  X[i,1]=Xe[i,1]    $ hold(i)=1..n ] ):
    Die:=subs(Diee, [  U[i,1]=Unom[i,1]    $ hold(i)=1..m ] ):
    print(Unquoted, "De = ", Die);

end_proc:


linearization:=proc()
local test;
begin

    // linearized system study
    symbolic_linearization():
    equilibrium_point_input():
    symbolic_linearization_in_Xe():
    linearized_eqns_compute(X, Y, U, "_lin"):
    repeat
      print();
      eqns_print(F_lin, G_lin):
      print();
      print(Unquoted, "a - Eigenvalues");
      print(Unquoted, "b - Reachability/Observability");
      print(Unquoted, "c - Transfer function analysis/Control design");
      print(Unquoted, "d - Linearized system time response");
      print(Unquoted, "e - Validation");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do eigenvalues(): break:
        of b do reach_obs_menu(): break:
        of c do transfer_function_analysis(): break:
        of d do linearized_time_response(): break:
        of e do validation(): break:
      end_case:
    until test=x end_repeat:

end_proc:


linearized_eqns_compute:=proc(X_formal, Y_formal, U_formal, suffix)
begin

    // Starting from matrix representation, write the corresponding
    // equations. Furthermore, starting from  the user variable
    // labels, form the lin and delta variables
    X_lin:=M(n,1):
    for i from 1 to n do
      X_lin[i,1]:=X_formal[i,1].text2expr(suffix):
    end_for:
    Y_lin:=M(p,1):
    for i from 1 to p do
```

```
      Y_lin[i,1]:=Y_formal[i,1].text2expr(suffix):
    end_for:

    F_lin:=Ae*(X_lin-Xe)+Be*(U_formal-Unom):
    G_lin:=Ce*(X_lin)+Die*(U_formal):


    delta_X:=M(n,1):
    for i from 1 to n do
      delta_X[i,1]:=text2expr("delta_").X_formal[i,1]:
    end_for:
    delta_Y:=M(p,1):
    for i from 1 to p do
      delta_Y[i,1]:=text2expr("delta_").Y_formal[i,1]:
    end_for:
    delta_U:=M(m,1):
    for i from 1 to m do
      delta_U[i,1]:=text2expr("delta_").U_formal[i,1]:
    end_for:
    delta_Unom:=M(m,1):
    for i from 1 to m do
      delta_Unom[i,1]:=0:
    end_for:

    F_delta:=Ae*delta_X+Be*delta_U:
    G_delta:=Ce*delta_X+Die*delta_U:

end_proc:

compute_R:=proc()
local i,j,k;
begin

    // Reachability matrix computing from all inputs and relative rank
    R:=M(n, n*m):
    for i from 0 to n-1 do
      R_block:=Ae^i*Be:
      for j from 1 to m do
        for k from 1 to n do
          R[k, i*m+j]:=R_block[k, j]:
        end_for:
      end_for:
    end_for:
    rank_R:=linalg::rank(R):
    print(Unquoted, "R = ", R);
    print(Unquoted, "rank(R)=" .expr2text(rank_R));
    if rank_R<n
      then print(Unquoted, "Not completely reachable system");
      else print(Unquoted, "Completely reachable system");
    end_if:

end_proc:


compute_O:=proc()
local i,j,k;
begin

    // Observability matrix computing from all outputs and relative rank
    // WARNING: we can not use O as identifier since it is a MuPAD keyword
    // We shall use Ob
    Ob:=M(p*n, n):
```

```
    for i from 0 to n-1 do
      Ob_block:=Ce*Ae^i:
      for j from 1 to p do
        for k from 1 to n do
          Ob[i*p+j, k]:=Ob_block[j, k]:
        end_for:
      end_for:
    end_for:
    rank_Ob:=linalg::rank(Ob):
    print(Unquoted, "O = ", Ob);
    print(Unquoted, "rank(O)=" .expr2text(rank_Ob));
    if rank_Ob<n
      then print(Unquoted, "Not completely observable system");
      else print(Unquoted, "Completely observable system");
    end_if:

end_proc:

total_r_o:=proc()
begin

    // Reachability and observability of the whole system
    compute_R():
    compute_O():

end_proc:


U_i_choice:=proc()
local test;
begin

    // Input choice for which reachability has to be checked
    repeat
      print(Unquoted, "Input vector U = ", U);
      print();
      input("Type in the index of the desired input variable:", i_reach):
      print();
      print(Unquoted, "Reachability study from: ");
      print(U[i_reach, 1]);
      input("Are these data right ? (y/n): ", test):
    until test=y end_repeat:
end_proc:


R_i_compute:=proc()
local i,j;
begin

    // Reachability matrix computing from one chosen input and relative rank
    R_i_reach:=M(n, n):
    Be_i_reach:=M(n, 1):
    for j from 1 to n do
      Be_i_reach[j, 1]:=Be[j, i_reach]:
    end_for:
    for i from 0 to n-1 do
      R_i_reach_col:=Ae^i*Be_i_reach:
        for j from 1 to n do
          R_i_reach[j, i+1]:=R_i_reach_col[j, 1]:
        end_for:
    end_for:
    rank_R:=linalg::rank(R_i_reach):
```

```
    print(Unquoted, "R = ", R_i_reach);
    print(Unquoted, "rank(R)=" .expr2text(rank_R));
    if rank_R<n
      then print(Unquoted, "Not completely reachable system from ".expr2text(U[i_reach, 1]));
      else print(Unquoted, "Completely reachable system from ".expr2text(U[i_reach, 1]));
    end_if:

end_proc:


single_r:=proc()
local test;
begin

    // Single input reachability procedure
    repeat
      U_i_choice():
      R_i_compute():
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Reachability from another input");
      print(Unquoted, "x - exit");
      print();
      input("",test);
    until test=x end_repeat;

end_proc:


Y_i_choice:=proc()
local test;
begin

    // Output choice for which observability has to be checked
    repeat
      print(Unquoted, "Output vector Y = ", Y_lin);
      print();
      input("Type in the index of the desired output variable:", i_obs):
      print();
      print(Unquoted, "Observability study from: ");
      print(Y_lin[i_obs, 1]);
      input("Are these data right ? (y/n): ", test):
    until test=y end_repeat:

end_proc:


Ob_i_compute:=proc()
local i,j;
begin

    // Observability matrix computing from one chosen output and relative rank
    Ob_i_obs:=M(n, n):
    Ce_i_obs:=M(1, n):
    for j from 1 to n do
      Ce_i_obs[1, j]:=Ce[i_obs, j]:
    end_for:
    for i from 0 to n-1 do
      Ob_i_obs_col:=Ce_i_obs*Ae^i:
        for j from 1 to n do
          Ob_i_obs[i+1, j]:=Ob_i_obs_col[1, j]:
        end_for:
```

```
    end_for:
    rank_Ob:=linalg::rank(Ob_i_obs):
    print(Unquoted, "O = ", Ob_i_obs);
    print(Unquoted, "rank(O)=" .expr2text(rank_Ob));
    if rank_Ob<n
      then print(Unquoted, "Not completely observable system from ".expr2text(Y_lin[i_obs, 1]));
      else print(Unquoted, "Completely observable system from ".expr2text(Y_lin[i_obs, 1]));
    end_if:

end_proc:



single_o:=proc()
local test;
begin

     // Single output observability procedure
    repeat
      Y_i_choice():
      Ob_i_compute():
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Observability from another input");
      print(Unquoted, "x - exit");
      print();
      input("",test);
    until test=x end_repeat;


end_proc:


single_r_o:=proc()
local test;
begin

    // Single reachability/observability options menu
    repeat
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Single reachability");
      print(Unquoted, "b - Single observability");
      print(Unquoted, "x - exit");
      print();
      input("",test);
      case test
        of a do single_r(): break;
        of b do single_o(): break;
      end_case;
    until test=x end_repeat;


end_proc:


reach_obs_menu:=proc()
local test;
begin

    // Total reachability (observability) or single input reachability
    // (single output observability) options menu
```

```
    repeat
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - From all inputs/outputs");
      print(Unquoted, "b - From single input/output");
      print(Unquoted, "x - exit");
      print();
      input("",test);
      case test
        of a do total_r_o(): break;
        of b do single_r_o(): break;
      end_case;
    until test=x end_repeat;

end_proc:
```

7.3 Example Application

To find the equilibrium points (s_e, x_e such that $\frac{ds}{dt} = 0, \frac{dx}{dt} = 0$), we set the input d to a nominal value $\overline{d} = 0.2\ day^{-1}$ in the example of section 6.3; hence, we have to solve the following nonlinear system:

$$\begin{cases} -\frac{1}{0.6}\frac{s}{2+s}x - 0.2s + 10 * 0.2 = 0 \\ \frac{s}{2+s}x - 0.02x - 0.2x = 0 \end{cases}$$

This system has three equilibrium points:(s,x)= (0.56,5.15), (-2,0), (10,0); only one has physical meaning (non-negative, non-zero):

$$\begin{cases} s_e = 0.56\ mg/l \\ x_e = 5.15\ mg/l \end{cases}$$

By calculation, we obtain the symbolic matrices which are then evaluated at the equilibrium point:

$$A = \frac{\partial f(X,U)}{\partial X}|_{(X_e,\overline{U})} = \begin{bmatrix} -2.82 & -0.36 \\ 1.57 & 0 \end{bmatrix}, \quad B = \frac{\partial f(X,U)}{\partial U}|_{(X_e,\overline{U})} = \begin{bmatrix} 9.44 \\ -5.15 \end{bmatrix},$$

$$C = \frac{\partial g(X,U)}{\partial X}|_{(X_e,\overline{U})} = \begin{bmatrix} 1 & 0 \end{bmatrix}, \quad D = \frac{\partial g(X,U)}{\partial U}|_{(X_e,\overline{U})} = 0$$

where

$$u = d, \ \ U = \overline{d} = 0.2, \ \ X = \begin{bmatrix} s \\ x \end{bmatrix},$$

$$X_e = \begin{bmatrix} 0.56 \\ 5.15 \end{bmatrix},$$

Hence, we obtain:

$$\begin{cases} \frac{dX}{dt} = \begin{bmatrix} -2.82 & -0.36 \\ 1.57 & 0 \end{bmatrix} X + \begin{bmatrix} 9.44 \\ -5.15 \end{bmatrix} u \\ \\ y = \begin{bmatrix} 1 & 0 \end{bmatrix} X \end{cases}$$

The local stability of the equilibrium point can be obtained by finding the eigenvalues of A, *i.e.,* the roots of $det[\lambda I - A] = 0$; those are equal to [-0.22,-2.6], implying that the point $X_e(0.56, 5.15)$ is locally stable. The reachability (R) and the observability (O) matrices are

$$R = \begin{bmatrix} 9.44 & -24.74 \\ -5.15 & 14.84 \end{bmatrix}$$

$$O = \begin{bmatrix} 1 & 0 \\ -2.81 & -0.36 \end{bmatrix}$$

The rank of R is 2 *i.e.,* the model is reachable. The rank of O is 2 *i.e.,* the model is observable.

8
Transfer Function Analysis

8.1 Theoretical Basis

Starting from the matrices A,B,C,D, by Laplace rules, we can get the transfer function representation of our system:

$$H(s) \quad = \quad C(sI - A)^{-1}B + D$$

We notice that for a MIMO (Multi Input Multi Output) system, H(s) is a rational matrix of dimension p*m. Let us consider an element $h(s) = H[i, j]$; h(s) is the transfer function between the j–th input and the i–th output. The external stability (i.e. I/O stability) of the open loop linearized model can be checked finding the poles of h(s). Once poles and zeros of h(s) are available, we can write the Zero–Pole form:

$$h(s) \quad = \quad k1\frac{(s - z_1)\dots(s - z_m)}{(s - p_1)\dots(s - p_n)}$$

H(s) in the above form implies that, with the Laplace transformation of the input U(s), the Laplace transformation of the output can be obtained with a simple multiplication:

$$Y(s) = H(s)U(s)$$

From h(s) we can also obtain the frequency response of the system:

$$H(\omega) = H(s = j\omega)$$

8.2 MuPAD Code

The procedures used are:

- transfer_function_analysis()
- compute_H()
- h_choice()

- h_study(); see section 16.6

```
transfer_function_analysis:=proc()
local test;
begin

    // MIMO transfer function H computing and analysis
    compute_H():
    repeat
      h_choice():
      h_study(h(s)):
      print(H);
      print();
      print(Unquoted, "a - Another transfer function study");
      print(Unquoted, "x - Exit");
      input("",test):
    until test=x end_repeat:

end_proc:


compute_H:=proc()
begin

    // Transfer function matrix computing. Since we consider MIMO
    // systems, we have a matrix of transfer functions H:
    //                 H= Ce*(s*Id-Ae)^(-1)*Be+Die
    // H is a p*m matrix (p=dim(Y), m=dim(U))

    Id:=M(n,n):
    for i from 1 to n do
      Id[i,i]:=1:
    end_for:
    SI_A_1:=1/(s*Id-Ae):

    // H1 is the unsimplified version of H
    H1:=Ce*SI_A_1*Be+Die:

    H:=M(p, m):
    // numer and denom function are useful to get a simplified
    // form of H(s)
    for i from 1 to p do
      for j from 1 to m do
        H[i,j]:=numer(H1[i,j])/denom(H1[i,j]):
      end_for:
    end_for:

end_proc:


h_choice:=proc()
begin

    // Choice of a SISO transfer function h(s) from a MIMO transfer
    // function H
    repeat
      print();
      print(Unquoted, "Transfer function matrix:");
      print(Unquoted, "H = ", H);
      print();
```

```
      print(Unquoted, "Choose one of the transfer function of the matrix");
      print();
      input("row index: ", row_index):
      print();
      input("column index: ", col_index):
      if row_index>p or col_index>m then
        print();
        print("WARNING: Indices out of ranges, please enter values again");
      end_if:
    until row_index<=p and col_index<=m end_repeat;
    print();
    h(s):=H[row_index,col_index]:
    print(Unquoted, "h(s) = ", h(s));

end_proc:
```

8.3 Example Application

In our example we get:

$$
\begin{aligned}
h(s) &= C(sI-A)^{-1}B+D \\
&= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} s+2.82 & +0.36 \\ -1.57 & s \end{bmatrix}^{-1} \begin{bmatrix} 9.44 \\ -5.15 \end{bmatrix} \\
&= \frac{6.01s+1.18}{0.64s^2+1.8s+0.36}
\end{aligned}
$$

Note that now s is a variable in the complex plane (do not confuse it with the substrate).
h(s) poles are equal to the eigenvalues of the matrix A; the zero of h(s) is located at $s=-0.2$ and the gain h(0) is 3.28. We can thus rewrite h(s) in the zero–pole form:

$$
h(s) = 9.38\frac{s+0.2}{(s+0.22)(s+2.6)}
$$

9
Frequency Response Plots

9.1 Theoretical Basis

From h(s) we can obtain the frequency response of the system:

$$H(\omega) = h(s = j\omega)$$

From the frequency response we can plot the following diagrams:

Bode $|H(\omega)|_{dB}$ versus ω and $phase(H(\omega))$ versus ω

Nyquist $Im(H(\omega))$ versus $Re(H(\omega))$

Nichols $|H(\omega)|_{dB}$ versus $phase(H(\omega))$

9.2 MuPAD Code

The procedures used are:

- plots(); see section 16.6.2

9.3 Example Application

From the frequency response of the system

$$H(\omega) = h(s = j\omega) \quad = \quad \frac{6.01j\omega + 1.18}{1.8j\omega - 0.64\omega^2 + 0.36}$$

we can plot:

- figure 9.1,9.2: Bode diagram
- figure 9.3: Nyquist diagram
- figure 9.4: Nichols diagram

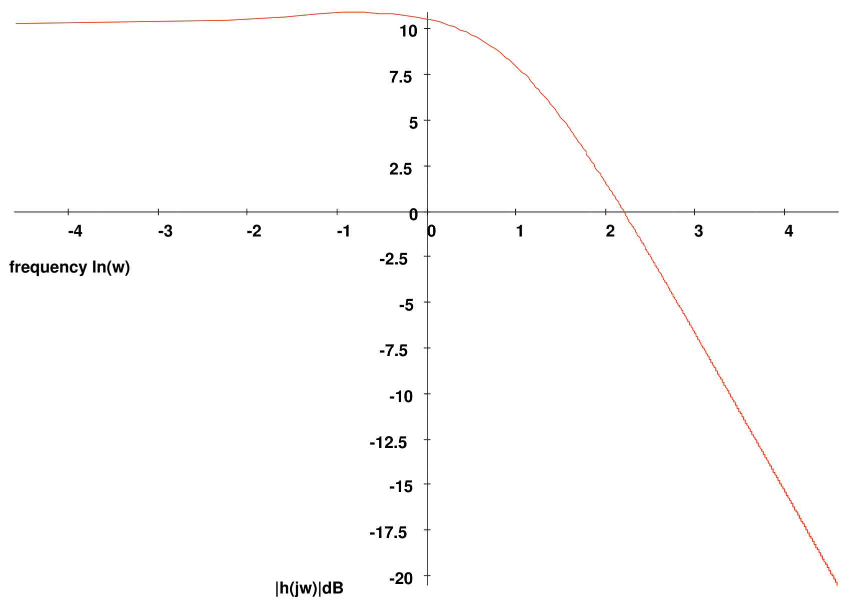


Figure 9.1: Bode amplitude plot

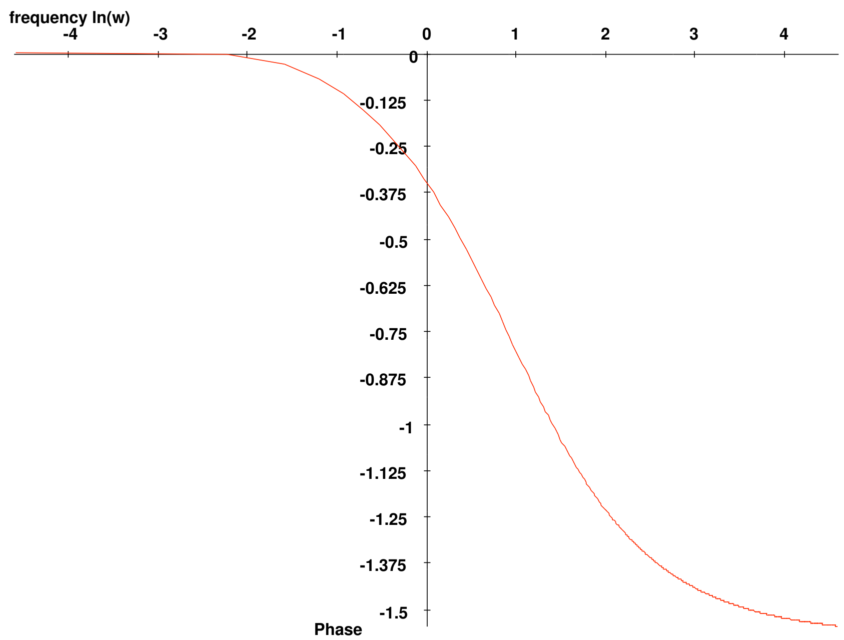


Figure 9.2: Bode phase plot

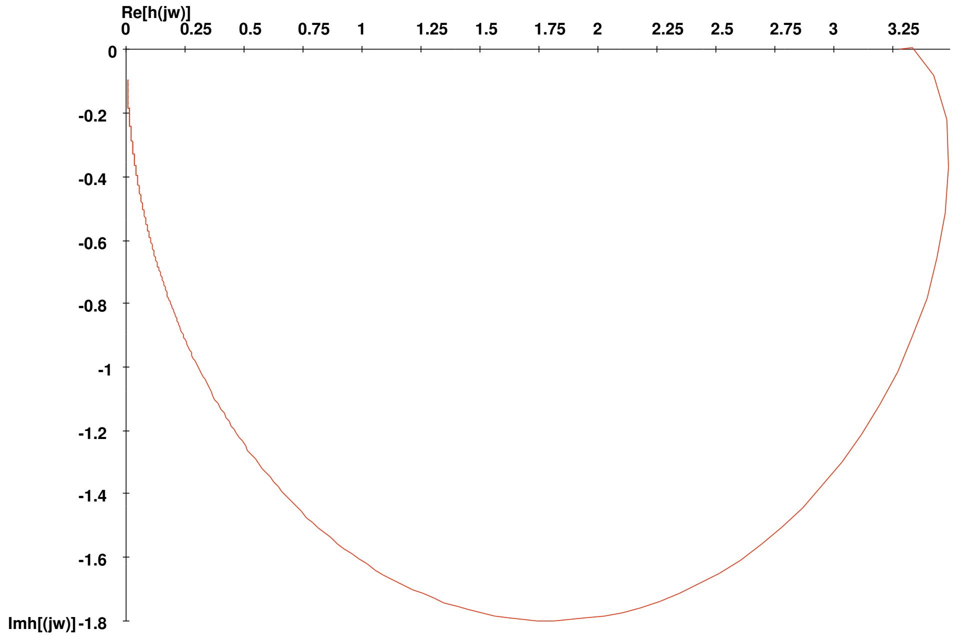


Figure 9.3: Nyquist plot

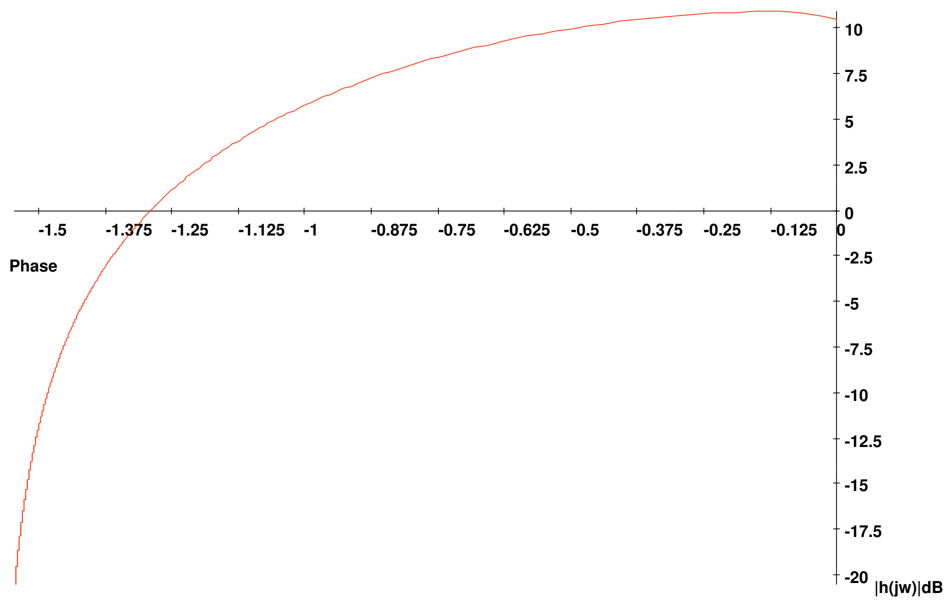


Figure 9.4: Nichols plot

10

Control Design – Stability Analysis Tools

10.1 Theoretical Basis

The previous chapters showed how to get a linearized system from a nonlinear one and how to have a transfer function $h(s)$ from the former. For a given $h(s)$ (*i.e.,* chosen an input and an output), we can try to build a controller ($c(s)$). Figure 10.1 shows the usual feedback scheme. Useful tools to design controllers are:

Root locus gives the closed loop pole trajectories as a function of the feedback gain k. Root loci are used to study the effects of varying feedback gains on closed loop pole locations. The closed loop poles are the roots of

$$q(s) = 1 + kc(s)h(s)$$

Gain and phase margins allow to get the stability degree of the closed loop system, based on the open loop frequency plot. The gain margin is the amount of gain increase required to make the loop gain unity at the frequency where the phase angle is $-180°$. In other words, the gain margin is $|1/g|dB$ if g is gain at the $-180°$ phase frequency. Similarly, the phase margin is the difference between the phase of the response and $-180°$ when the loop gain is 1.

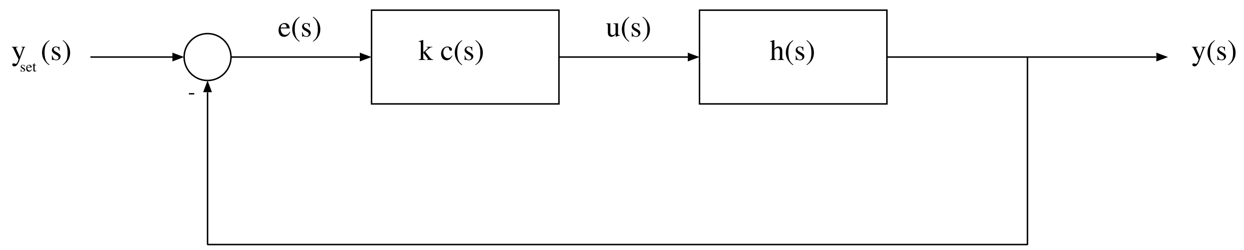


Figure 10.1: Feedback control scheme

10.2 MuPAD Code

The procedures used are:

- control_menu()
- stability_analysis_tools()

- root_locus()
- margins()
- gain_margin()
- phase_margin()

```
control_menu:=proc()
local test;
begin

    // Control menu
    repeat
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Stability analysis tools");
      print(Unquoted, "b - Controllers");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do stability_analysis_tools(h(s)): break:
        of b do controllers(): break:
      end_case:
    until test=x end_repeat:

end_proc:

root_locus:=proc(h_formal)
local denh, numh, numh_deg, locuseqtn, k_in, k_fin, k_step, i, k, k1,
data, data_denh, data_numh, dim_data, dim_elem_data, j, re_data, im_data, re_data_denh, im_data_denh,re_data_numh, im_data_numh, plotpoints,
plotpoints_numh, plotpoints_denh;
begin

    // Roots locus plot from k_in to k_fin and stepsize k_step
    denh:=denom(h_formal):
    numh:=numer(h_formal):
    numh_deg:=degree(numh):
    data_denh:=float(roots_finding(denh)):
    locuseqtn:=expand(denh+k*numh);
    print(Unquoted, "Root locus equation:");
    print(locuseqtn);
    input("Enter starting value of k:", k_in);
    print();
    input("Enter final value of k:", k_fin);
    print();
    input("Enter stepsize for k:", k_step);
    print():
    i:=0;
    for k1 from k_in to k_fin step k_step do
      locuseval:=subs(locuseqtn,k=k1);
      data[i]:=float(roots_finding(locuseval));
      if nops(data[i])<degree(locuseval) then
        if i>0 then
          data[i]:=data[i-1]:
        else
          data[i]:=array(1..degree(locuseval)):
          for l from 1 to degree(locuseval) do
            data[i][l]:=0:
          end_for:
```

```
        end_if:
      end_if:
      i:=i+1;
    end_for;
    dim_data:=nops(data);
    dim_elem_data:=nops(data[0]);
    for i from 1 to dim_data do
      for j from 1 to dim_elem_data do
        re_data[dim_elem_data*(i-1)+j]:=Re(op(data[i-1], j));
        im_data[dim_elem_data*(i-1)+j]:=Im(op(data[i-1], j));
      end_for;
    end_for;
   // h_formal poles storing loops
   for j from 1 to nops(data_denh) do
      re_data_denh[j]:=Re(op(data_denh, j));
      im_data_denh[j]:=Im(op(data_denh, j));
    end_for:

    plotpoints:=[point(re_data[i], im_data[i]) $ hold(i)=1..nops(re_data)];
    plotpoints_denh:=[point(re_data_denh[i], im_data_denh[i]) $ hold(i)=1..nops(re_data_denh)];
    if numh_deg=0
      then  plot2d(Scaling=UnConstrained, Labeling=TRUE, Axes=Origin,
                   AxesOrigin=[0, 0], Labels=["Re[s]", "Im[s]"],
                   BackGround=[1.0, 1.0, 1.0], ForeGround=[0.0, 0.0, 0.0],
                   [Mode=List, plotpoints, PointStyle=FilledCircles],
                   [Mode=List, plotpoints_denh, PointStyle=Squares,
                   Color=[Flat, [0.0, 0.0, 1.0]]]):
      else data_numh:=float(roots_finding(numh)):
           for j from 1 to nops(data_numh) do
             re_data_numh[j]:=Re(op(data_numh, j));
             im_data_numh[j]:=Im(op(data_numh, j));
           end_for:
           plotpoints_numh:=[point(re_data_numh[i], im_data_numh[i]) $ hold(i)=1..nops(re_data_numh)];
           plot2d(Scaling=UnConstrained, Labeling=TRUE, Axes=Origin,
                  AxesOrigin=[0, 0], Labels=["Re[s]", "Im[s]"],
                  BackGround=[1.0, 1.0, 1.0], ForeGround=[0.0, 0.0, 0.0],
                  [Mode=List, plotpoints, PointStyle=FilledCircles],
                  [Mode=List, plotpoints_denh, PointStyle=Squares,
                  Color=[Flat, [0.0, 0.0, 1.0]]],
                  [Mode=List, plotpoints_numh, PointStyle=Squares,
                  Color=[Flat, [0.0, 0.0, 1.0]]]):
    end_if:

end_proc:


gain_margin:=proc(h_formal)
  local w_set,w_g,g_m,i;
begin

    // Compute the gain margin of a transfer function
    if degree(numer(imhw(w)))=0
      then return(+INFINITY):
    end_if:
    w_set_g:=float(numeric::bairstow(numer(imhw(w)))):
    if w_set_g=FAIL then
      print(Unquoted, "Not able to solve");
      return(FAIL):
    end_if:
    i:=0:
    repeat
      i:=i+1:
```

```
    until i>nops(w_set_g) or
          (Re(op(w_set_g,i))>0.01 and domtype(op(w_set_g,i))=DOM_FLOAT
           and float(subs(rehw(w),w=op(w_set_g,i)))<0)
    end_repeat:
    if i>nops(w_set_g) then
      return(+INFINITY);
    end_if;
    w_g:=op(w_set_g,i):

    g_m(w):=1/amplhw(w):
    subs(g_m(w),w=w_g);

end_proc:


phase_margin:=proc(h_formal)
  local w_set,w_p,ph_m;
begin

     // Compute the phase margin of a transfer function
     amplhw2(w):=amplhw(w)^2:
     w_set_p:=float(numeric::bairstow(numer(amplhw2(w))-denom(amplhw2(w)))):
     if w_set_p=FAIL then
       print(Unquoted, "Not able to solve");
       return(FAIL):
     end_if:
     i:=0:
     repeat
       i:=i+1:
     until i>nops(w_set_p) or
           (Re(op(w_set_p,i))>0.01 and domtype(op(w_set_p,i))=DOM_FLOAT)
     end_repeat:
     if i>nops(w_set_p) then
       return(FAIL)
     end_if;

     w_p:=op(w_set_p,i):
     ph_m(w):=180+phase(h_formal):
     subs(ph_m(w),w=w_p);

end_proc:


margins:=proc(h_formal)
  local g_m1,g_m_db,p_m1;
begin

    // gain and phase margins
    // amplitude(h_formal) ==> amplitude of h(Iw) and rehw,imhw globals
    amplhw(w):=amplitude(h_formal):
    phasehw(w):=phase(h_formal):

    g_m1:=float(gain_margin(h_formal)):
    p_m1:=float(phase_margin(h_formal)):
    g_m_db:=float(20*ln(g_m1)/ln(10)):

    print():
    print(Unquoted, "The gain margin is (dB)".expr2text(g_m_db));
    print(Unquoted, "The phase margin is ".expr2text(level(p_m1)));

end_proc:
```

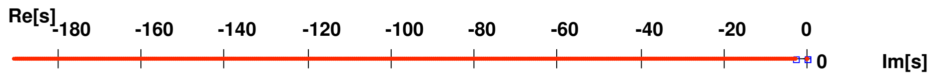


Figure 10.2: Root Locus $(k = 0 \dots 20)$

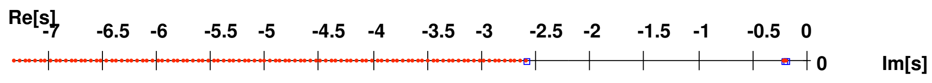


Figure 10.3: Root Locus $(k = 0 \dots 0.5)$

```
stability_analysis_tools:=proc(h_formal)
local test;
begin

    // Stability analysis tools menu
    repeat
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Root locus");
      print(Unquoted, "b - Gain/Phase margins");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do root_locus(h_formal): break:
        of b do margins(h_formal): break:
      end_case:
    until test=x end_repeat:

end_proc:
```

10.3 Example Application

The root locus equation with $c(s) = 1$ is:

$$q(s) = 1 + kh(s) = 1.18k + 1.8s + 6.01ks + 0.64s^2 + 0.36$$

Figure 10.2 is the plot of $q(s)$ roots for k varying from 0 to 20 (step=0.01); Figure 10.3 is the root locus for k varying between 0 and 0.5 (step=0.005).
The gain margin is plus infinity, the phase margin is 106 degrees.

11

Control Design – Controllers

11.1 Theoretical Basis

We have considered some standard controllers based on the classic scheme shown in figure 11.1; also we have analysed a control based on the internal model control (IMC) scheme (see figure 11.2).

PID have a fixed structure:

$$c(s) = K_p(1 + T_d s + \frac{1}{T_i s})$$

We will fix the parameters using the Nyquist criterion. The user can choose the design frequency (ω_0) and can impose the phase (the difference from $-180°$) and the amplitude of the forward chain frequency response at $\omega = \omega_0$. Furthermore, because of the PID integral action, this feedback controller yields both asymptotic tracking and constant disturbance rejection.

Direct design When a fixed controller structure is not required, after setting the closed loop characteristics, we can try to design a suitable controller whenever possible.Starting from the open loop transfer function h(s), a desired closed loop transfer function $w(s)$ is obtained by means of the controller:

$$c(s) = \frac{1}{h(s)}\frac{w(s)}{1 - w(s)}$$

The choice of $w(s)$ has some constraints depending on $h(s)$.

Trial and error/User controller The user has the possibility to choose a controller $c(s)$ and to check the controller, the forward chain and the closed loop behavior obtained.

Robust control This is based on the IMC structure (figure 11.2) introduced as an alternative to the classic feedback structure. Its main advantage is that, if the process is stable, the closed loop stability is assured simply by choosing a stable IMC controller. We have used a two–step design procedure. In the first step the controller $\tilde{c}(s)$ is selected to yield a "good" system response for the input of interest (we have considered only the case of a step input). In the second step $\tilde{c}(s)$ is augmented by a low-pass filter $f(s)$ to achieve robust stability and robust performance. The IMC controller will be:

$$c(s) = \tilde{c}(s)f(s)$$

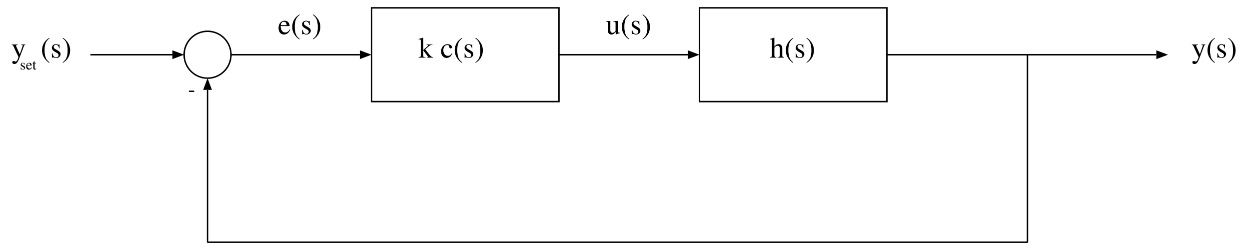


Figure 11.1: Feedback control scheme

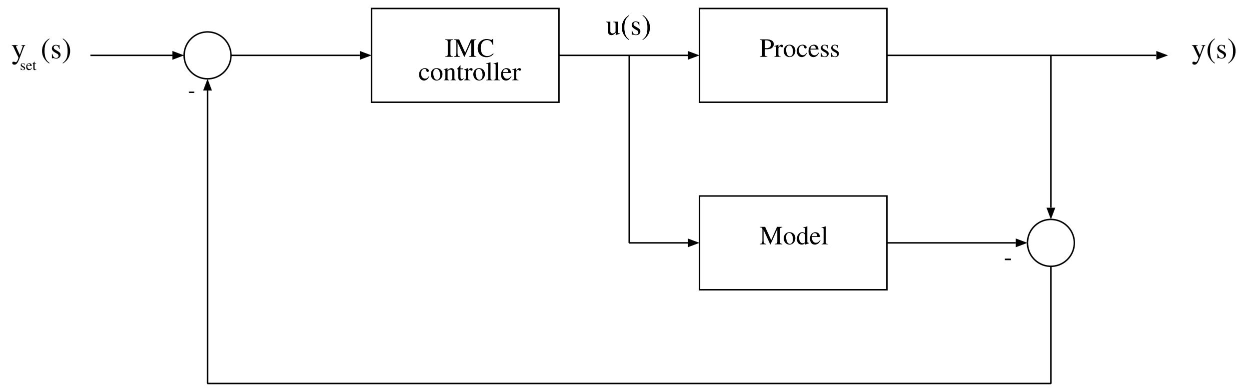


Figure 11.2: IMC control scheme

11.2 MuPAD Code

The procedures used are:

- controllers()
- PID_nyquist_plot()
- PID_nyquist()
- h_analysis()
- W_input()
- direct_design()
- user_controller()
- robust_control()
- controller_analysis()
- controlled_forward_chain_analysis()
- closed_loop_analysis()
- control_verify()
- plots(); see section 16.6.2
- pole_zero(); see section 16.6.2

```
PID_nyquist_plot:=proc(h_formal)
local w_in, w_fin;
begin

    // Nyquist plot for a forward chain with PID controller
    // The frequency of design is also plotted
    real_imaginary(h_formal):

    print(Unquoted, "Nyquist plot: Enter the initial value for w (w>0)");
    w_in:=input(""):
    print(Unquoted, "Nyquist plot: Enter the final value for w");
    w_fin:=input(""):
    plot2d(Scaling=UnConstrained, Labeling=TRUE, Axes=Origin,
    AxesOrigin=[0, 0], Labels=["Re[h(jw)]", "Imh[(jw)]"],
    BackGround=[1.0, 1.0, 1.0],
    ForeGround=[0.0, 0.0, 0.0],
    [Mode=Curve, [float(rehw(w)), float(imhw(w))], w=[w_in, w_fin],
    Grid=[50], Smoothness=[20]],
    [Mode=List, [point(float(subs(rehw(w), w=w0)), float(subs(imhw(w), w=w0)))],
    Color=[Flat, [0.0, 0.0, 1.0]]]);

end_proc:


PID_nyquist:=proc()
local amplhw, amplhw0, phasehw, phasehw0, des_amp, des_phase,
des_phase_rad, phasehw0_rad, K_p, T_i, T_d, pid, new_h;
begin

    // PID tuning with Nyquist criteria
    input("Enter the frequency of design w0:", w0):
    amplhw:=amplitude(h(s)):
    amplhw0:=float(subs(amplhw, w=w0)):
    phasehw:=180.0+phase(h(s)):
    phasehw0:=float(subs(phasehw, w=w0)):
    PID_nyquist_plot(h(s)):
    print(Unquoted,
      "Open loop amplitude at w0=" .expr2text(w0) .": " .expr2text(amplhw0));
    print(Unquoted,
      "Open loop phase at w0=" .expr2text(w0) .": " .expr2text(level(phasehw0)));
    print():
    print(Unquoted,
      "Enter the desired forward chain amplitude at w0=".expr2text(w0));
    input("", des_amp):
    print();
    print(Unquoted,
      "Enter the desired forward chain phase (degree) at w0=".expr2text(w0));
     input("", des_phase):
    print();
    des_phase_rad:=des_phase*PI/180:
    phasehw0_rad:=phasehw0*PI/180:
    K_p:=float(des_amp/amplhw0*cos(des_phase_rad-phasehw0_rad)):
    T_i:=float((tan(des_phase_rad-phasehw0_rad)
      +sqrt(2+(tan(des_phase_rad-phasehw0_rad))^2))/w0):
    T_d:=float(T_i/2):
    print(Unquoted, "PID parameters:");
    print(Unquoted, "Kp=".expr2text(level(K_p)));
    print(Unquoted, "Ti=".expr2text(level(T_i)));
    print(Unquoted, "Td=".expr2text(level(T_d)));
    pid(s):=K_p*(1+1/(T_i*s)+T_d*s/(1+T_d*s/10));
    C(s):=pid(s):
```

```
    // new_h(s) is the new forward chain (PID * h(s))
    new_h(s):=expand(pid(s)*h(s)):
    // print(Unquoted, "New forward chain:", new_h(s));
    PID_nyquist_plot(new_h(s)):

end_proc:


h_analysis:=proc()
local h_num, h_den, h_num_deg, h_den_deg, num_roots, den_roots;
begin

    // Open loop h(s) study to suggest conditions that the closed
    // loop W(s) has to satisfy in order to get a realizable controller
    // and a stable closed loop system
    h_num:=numer(h(s)):
    h_den:=denom(h(s)):
    h_num_deg:=degree(h_num):
    h_den_deg:=degree(h_den):
    excess_h:=h_den_deg-h_num_deg:
    print();
    print(Unquoted, "Realizability condition:"):
    print(Unquoted, "W(s) pole-zero excess should be >= ".expr2text(excess_h));
    if h_num_deg>0
      then num_roots:=roots_finding(h_num):
      else num_roots:={}:
    end_if:
    if h_den_deg>0
      then den_roots:=roots_finding(h_den):
      else den_roots:={} :
    end_if:
    print(Unquoted, "Stability condition:"):
    print(Unquoted, "h(s) zeros are: ");
    print(num_roots);
    print(Unquoted, "W(s) should contain the instable zeros (Re>=0) of h(s)");
    print(Unquoted, "h(s) poles are: ");
    print(den_roots);
    print(Unquoted,
      "W(s) should be equal 1 at the instable poles (Re>=0) of h(s)");

end_proc:


W_input:=proc()
local test, W_num_deg, W_den_deg, excess_W;
begin

    // Desired closed loop transfer function input
    repeat
      repeat
        repeat
          print(Unquoted,
            "Enter the desired closed loop transfer function W(s):");
          W(s):=input(""):
          print(W(s));
          print(Unquoted, "Are these data right ? (y/n)"):
          test:=input(""):
        until test=y end_repeat:
      W_num_deg:=degree(numer(W(s))):
      W_den_deg:=degree(denom(W(s))):
      if W_den_deg<W_num_deg
        then print(Unquoted, "WARNING: W(s) must be a causal transfer function");
```

```
      end_if:
      until W_den_deg>=W_num_deg end_repeat:
      excess_W:=W_den_deg-W_num_deg:
      if excess_W<excess_h
        then print(Unquoted,
               "WARNING: The pole-zero excess of W(s) must be >= "
               .expr2text(excess_h));
      end_if:
    until excess_W>=excess_h end_repeat:

end_proc:


direct_design:=proc()
begin

    // Direct design controller C(s) computing
    print(Unquoted, "Starting from the open loop transfer function h(s):=", h(s)):
    print(Unquoted,
    "a desired closed loop behaviour W(s) is obtained by means of the controller:"):
    print(Unquoted, "C(s):= 1/h(s)*W(s)/(1-W(s))"):
    h_analysis():
    W_input():
    C(s):=1/h(s)*W(s)/(1-W(s)):
    print(Unquoted, "The controller obtained is: ");
    print(Unquoted, "C(s)=",normal(C(s)));

end_proc:


robust_control:=proc(h_formal)
local i, j, hnum, hden, n_hnum, n_hden, z_hden, facthden, z_hnum, facthnum,
gain_correct, C_nom_1, C_nom, C, f, h0, hpz0, k, hpz;
begin

    // Robust control using Internal Model Control (IMC) approach
    // h(s) numerator, denominator and their degrees
    hnum(s):=numer(h_formal):
    hden(s):=denom(h_formal):
    n_hnum:=degree(hnum(s),s):
    n_hden:=degree(hden(s),s):

    // denominator in factorized form
    if n_hden>0 then
      z_hden:=roots_finding(hden(s)):   // denominator zeros
      if z_hden=FAIL then return(FAIL) end_if:
      print(Unquoted,
        "Warning: Robust control should not be used for instable h(s)"):
      print(Unquoted, "h(s) poles are:");
      print(z_hden);
      facthden(s):=(s-op(z_hden,1)):
      for i from 2 to n_hden do
        facthden(s):=facthden(s)*(s-op(z_hden,i));
      end_for;
    else facthden(s):=hden(s);
    end_if;

    // numerator in factorized form
    j:=0:
    if n_hnum>0 then
      z_hnum:=roots_finding(hnum(s)):   // numerator zeros
      if z_hnum=FAIL then return(FAIL) end_if:
```

```
      if Re(op(z_hnum,1))>0
        then facthnum(s):=(s+op(z_hnum,1)):
             j:=j+1:
        else facthnum(s):=(s-op(z_hnum,1)):
      end_if:
      for i from 2 to n_hnum do
        if Re(op(z_hnum,i))>0
          then facthnum(s):=facthnum(s)*(s+op(z_hnum,i)):
               j:=j+1:
          else facthnum(s):=facthnum(s)*(s-op(z_hnum,i)):
        end_if:
      end_for;
    else facthnum(s):=hnum(s);
    end_if;

    if j mod 2 = 0
      then gain_correct:=1:
      else gain_correct:=-1:
    end_if:

    // h_formal in pole-zero form: hpz(s)
    i:=-1:
    repeat
      i:=i+1:
    until (float(op(divide(hden(s),s-i),2))<>0.0
      and float(op(divide(hnum(s),s-i),2))<>0.0)
    end_repeat:

    hpz(s):=facthnum(s)/facthden(s);   // wrong gain !
    C_nom_1(s):=facthden(s)/facthnum(s);   // wrong gain !
    h0:=subs(h_formal,s=i):
    hpz0:=Re(subs(hpz(s), s=i));
    k:=abs(h0/hpz0);
    k:=float(k);
    C_nom(s):=(1/k)*C_nom_1(s)*gain_correct:    // this is the right C_nom(s)
    print(Unquoted,"The nominal controller is:");
    print(float(C_nom(s))):
    if n_hden-n_hnum = 0
      then f(s):=1/(1+a*s):
      else f(s):=1/(1+a*s)^(n_hden-n_hnum):
    end_if:
    C(s):=C_nom(s)*f(s):
    print(Unquoted, "The robust controller is:");
    print(Unquoted, "C(s)= ", C(s)):

end_proc:


user_controller:=proc()
local test;
begin

    // User defined controller
    repeat
      repeat
        print();
        print(Unquoted, "Enter the controller transfer function: ");
        C(s):=input(""):
        print(C(s));
      if degree(denom(C(s))) < degree(numer(C(s)))
        then print(Unquoted, "Warning: Improper transfer function");
      end_if:
```

```
      until degree(denom(C(s))) >= degree(numer(C(s))) end_repeat;
      print(Unquoted, "Are these data right ? (y/n)");
      input("",test);
    until test=y end_repeat:

end_proc:


controller_analysis:=proc()
local test, C_normal;
begin

    // Controller analysis menu
    repeat
      print();
      C_normal(s):=normal(C(s)):
      print(Unquoted, "C(s)=", C_normal(s));
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Plots");
      print(Unquoted, "b - Pole-zero representation");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do plots(C_normal(s)): break:
        of b do pole_zero(C_normal(s)): break:
       end_case:
    until test=x end_repeat:

end_proc:


controlled_forward_chain_analysis:=proc()
local test, L, L_normal;
begin

    // Controlled forward chain analysis menu
    L(s):=C(s)*h(s):
    repeat
      print();
      print(Unquoted, "L(s)=C(s)*h(s)=", L(s));
      L_normal(s):=normal(L(s)):
      print(Unquoted, "=", L_normal(s));
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Plots");
      print(Unquoted, "b - Pole-zero representation");
      print(Unquoted, "c - Stability analysis tools");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do plots(L_normal(s)): break:
        of b do pole_zero(L_normal(s)): break:
        of c do stability_analysis_tools(L_normal(s)): break:
       end_case:
    until test=x end_repeat:

end_proc:
```

```
closed_loop_analysis:=proc()
local test, W;
begin

   // closed loop analysis menu
   W(s):=normal(C(s)*h(s)/(1+C(s)*h(s))):
   repeat
      print();
      print(Unquoted, "W(s)=C(s)*h(s)/(1+C(s)*h(s))=", W(s));
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Plots");
      print(Unquoted, "b - Pole-zero representation");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do plots(W(s)): break:
        of b do pole_zero(W(s)): break:
       end_case:
    until test=x end_repeat:

end_proc:


control_verify:=proc()
local test;
begin

    // Controller, controlled forward chain and closed loop analysis menu
    repeat
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Controller analysis");
      print(Unquoted, "b - Controlled forward chain analysis");
      print(Unquoted, "c - Closed loop analysis");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do controller_analysis(): break:
        of b do controlled_forward_chain_analysis(): break:
        of c do closed_loop_analysis(): break:
      end_case:
    until test=x end_repeat:

end_proc:

controllers:=proc()
local test;
begin

    // Controllers menu
    repeat
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - PID");
      print(Unquoted, "b - Direct Design");
      print(Unquoted, "c - Robust Control");
      print(Unquoted, "d - User Controller/Trial and error");
      print(Unquoted, "x - Exit");
      print();
```

```
      input("",test);
      case test
        of a do PID_nyquist(): break:
        of b do direct_design(): break:
        of c do robust_control(h(s)): break:
        of d do user_controller(): break:
      end_case:
      if test<>x and test<>c
        then control_verify():
      end_if:
    until test=x end_repeat:

end_proc:
```

11.3 Example Application

With reference to the example of section 6.3 we have built some controllers:

PID We have made two PID controllers:

- with a choice of $\omega_0 = 1$, $|c(j\omega_0)h(j\omega_0)| = 3.4$ and $phase(c(j\omega_0)h(j\omega_0)) - (-180°) = 160°$, we get: $K_p = 1.011$, $T_i = 1.41$, $T_d = 0.7$.
- with a choice of $\omega_0 = 0.5$, $|c(j\omega_0)h(j\omega_0)| = 3.5$ and $phase(c(j\omega_0)h(j\omega_0)) - (-180°) = 171°$, we get: $K_p = 1$, $T_i = 2.82$, $T_d = 1.41$.

Direct design We have tried with two $w(s)$:

- $w(s) = \frac{1}{s+1}$; we get $c(s) = \frac{0.64s^2+1.8s+0.36}{6.01s^2+1.18s}$
- $w(s) = \frac{4}{(s+2)^2}$ we get $c(s) = \frac{2.56s^2+7.2s+1.44}{6.01s^3+25.22s^2+4.72s}$

Trial and error/User controller We have chosen a simple integrator

$$c(s) = \frac{1}{s}$$

and which leads to a closed loop transfer function

$$w(s) = \frac{6.01s + 1.18}{0.64s^3 + 1.8s^2 + 6.37s + 1.18}$$

Robust control We get:

$$\tilde{c}(s) = \frac{1}{9.38}\frac{(s + 0.22)(s + 2.6)}{(s + 0.2)}$$

$$f(s) = \frac{1}{(1 + as)^2}$$

$$c(s) = \frac{1}{9.38}\frac{(s + 0.22)(s + 2.6)}{(1 + as)^2(s + 0.2)}$$

the parameter a has to be chosen on line.

The third part will discuss more in detail controllers.

12
Linearized System Time Response

12.1 Theoretical Basis

It can be useful to study the state plot or the output plot for different kinds of inputs.

12.2 MuPAD Code

The procedures used are:

- linearized_time_response()
- tmp_set(); see section 16.2
- compute_response(); see section 16.4

```
linearized_time_response:=proc()
local test;
begin

    // Linearized and delta linearized options menu
    repeat
      print():
      print(Unquoted, "Linearized system:");
      eqns_print(F_lin, G_lin):
      print(Unquoted, "Delta linearized system:");
      eqns_print(F_delta, G_delta):
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Linearized system response");
      print(Unquoted, "b - Delta linearized system response");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do tmp_set(F_lin, G_lin, X_lin, Y_lin, U, Unom):
                compute_response():
                break:
        of b do tmp_set(F_delta, G_delta, delta_X, delta_Y, delta_U, delta_Unom):
                compute_response():
                break:
```

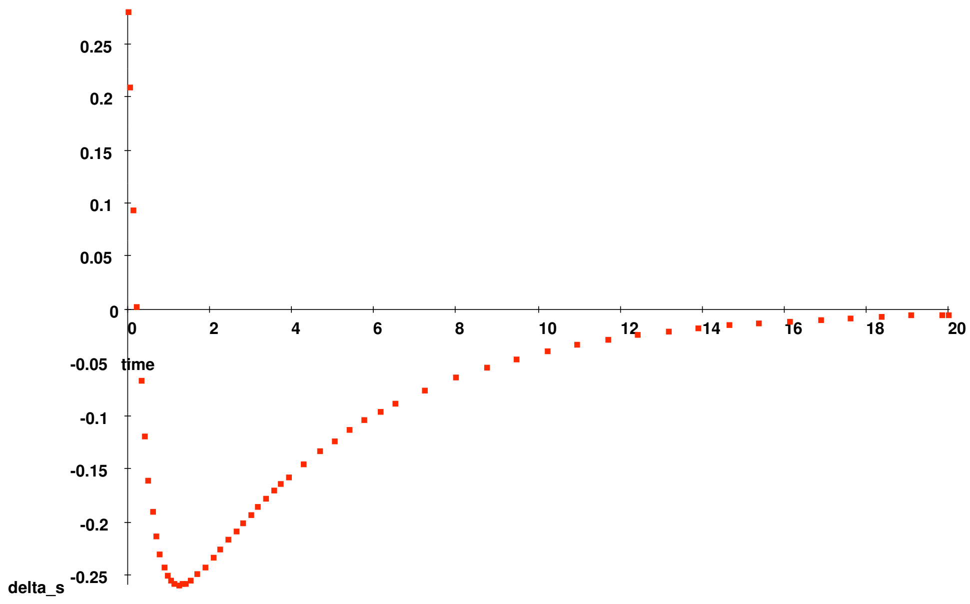


Figure 12.1: $\Delta s(t)$ free response plot

```
      end_case:
    until test=x end_repeat:

end_proc:
```

12.3 Example Application

For example we can plot the following diagrams:

- figure 12.1, 12.2: free response of the state for the linear system $(\Delta s(t), \Delta x(t))$,with initial conditions set to $s(0) = 0.84,\ x(0) = 7.725$.
- figure 12.3,12.4: free response of the state for the linear system $(s(t), x(t))$,with initial conditions set to $s(0) = 0.84,\ x(0) = 7.725$.
- figure 12.5, 12.6: step response of the state for the linear system $(s(t), x(t))$, with input amplitude set to $d = \overline{d} + 0.1$ $(\overline{d} = 0.2)$.

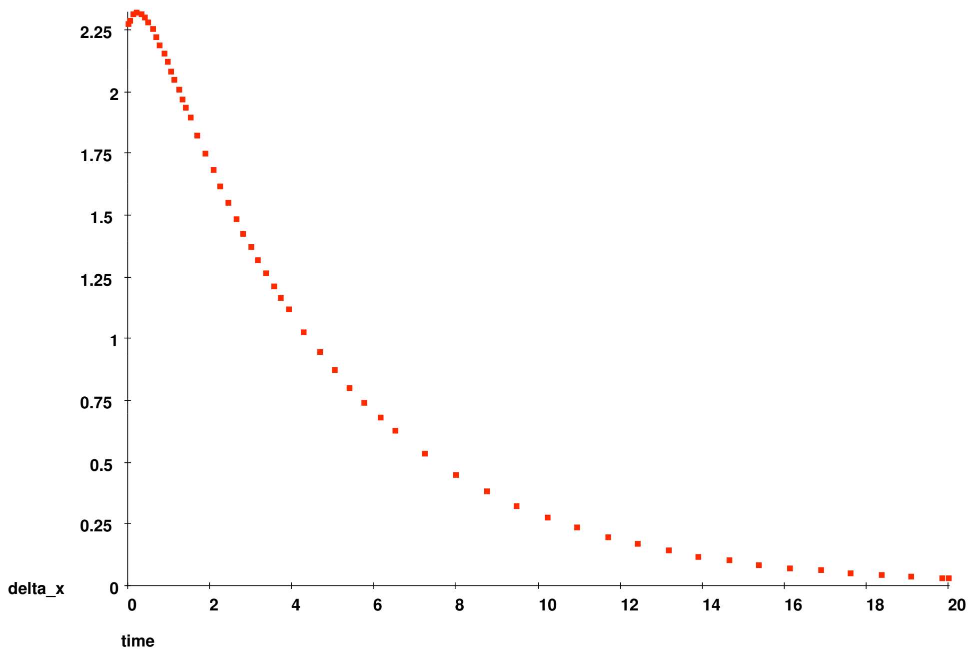


Figure 12.2: $\Delta x(t)$ free response plot

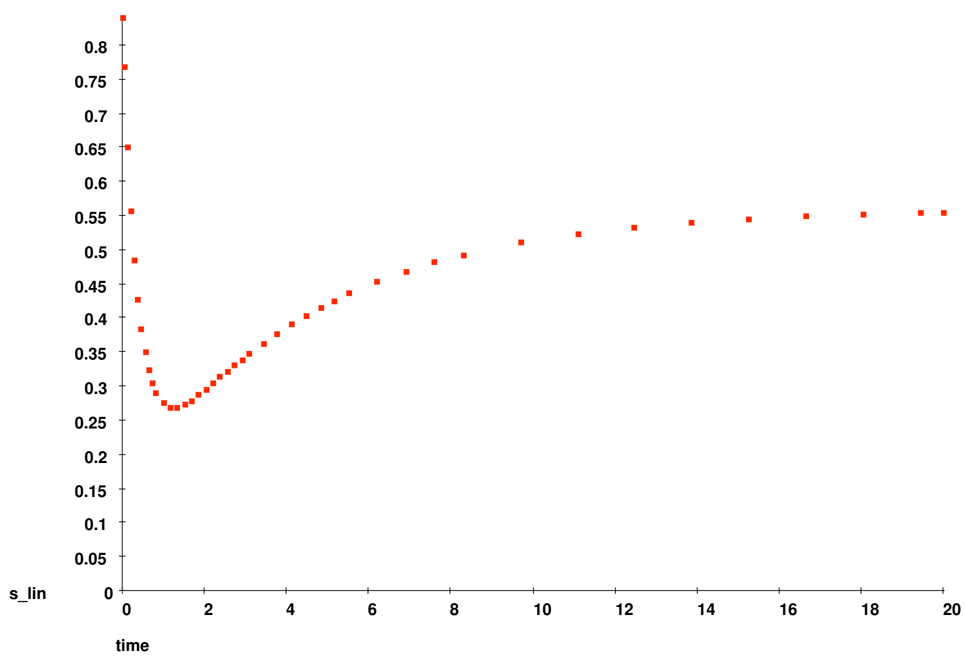


Figure 12.3: s(t) free response plot

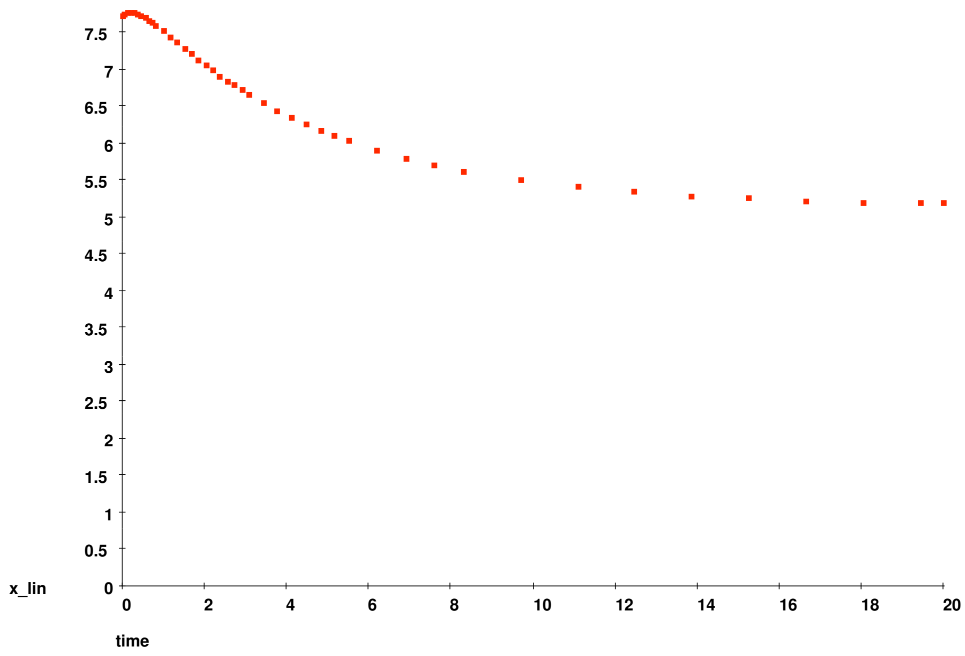


Figure 12.4: x(t) free response plot

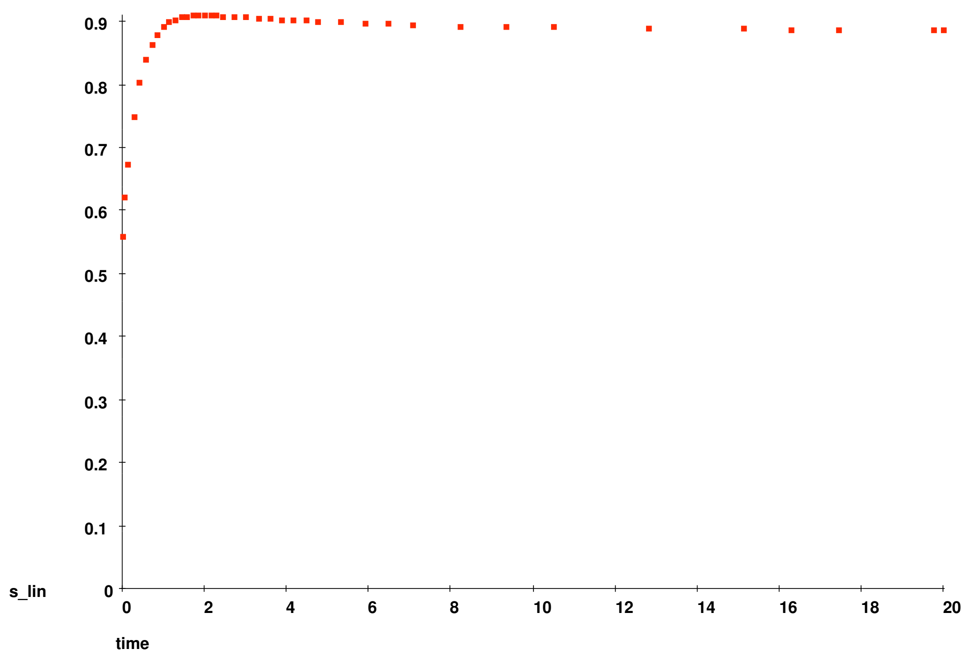


Figure 12.5: s(t) step response plot

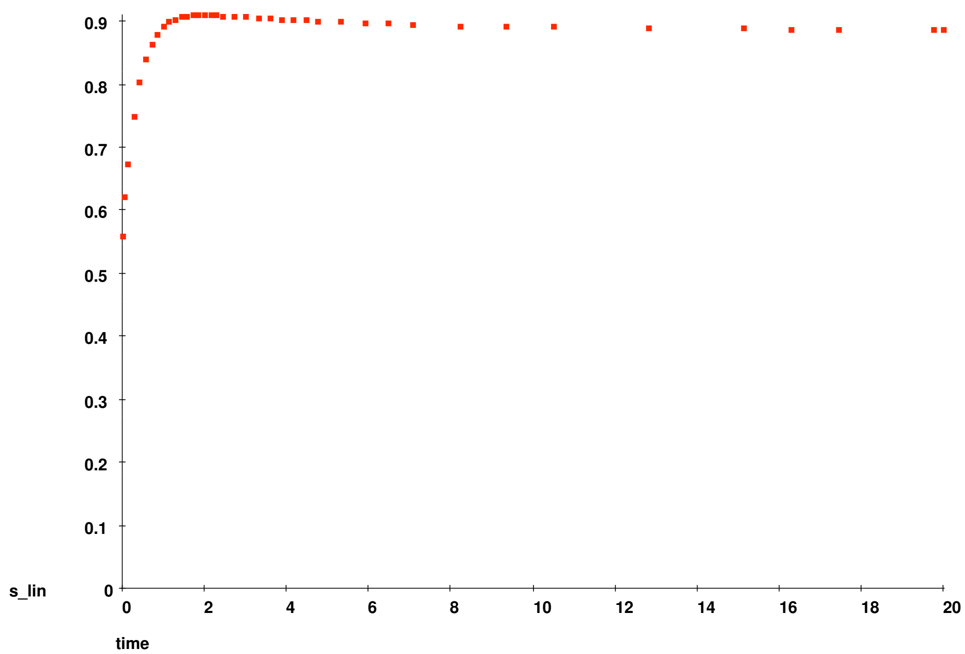


Figure 12.6: x(t) step response plot

13
Validation

13.1 Theoretical Basis

It is important to check if the linearized model is a good approximation of the nonlinear model. To verify this, we have to make some tests comparing the response of the nonlinear system with the response of the linearized system. We need both a quantitative validation and a qualitative validation. An appropriate measure is the mean square error:

$$\sqrt{\frac{1}{t_e - t_i} \sum_{t=t_i}^{t_e} (x(t) - x_{lin}(t))^2}$$

where x is the variable in the nonlinear system to be verified, x_{lin} is the corresponding linearized variable, t_i is the initial time and t_e denotes the time of the end of the simulation. A qualitative validation can be obtained by plotting x and x_{lin} together.

13.2 MuPAD Code

The procedures used are:

- validation()
- eqns_print(); see section 16.5
- tmp_set(); see section 16.2
- pre_set_parameters(); see section 16.4.1
- input_choice(); see section 16.4.2
- step_size_set()
- compute_response_1(); see section 16.4.3
- valid_response()
- valid_Xj_or_Yj()
- compute_data_Yj(); see section 16.4.4

- validation_plot()
- time_val()
- err_val()
- plot_val()

```
time_val:=proc()
begin

    // Setting of the time at which the nonlinear and linear system
    // have to be compared
    repeat
      print(Unquoted, "enter the validation starting time: ");
      input("",tv):
      if tv>t1 then
        print(Unquoted,
        "the validation time must be less or equal to the final time");
      end_if:
    until tv<=t1 end_repeat:

    //   start time
    i:=1:
    while l_data_X[i][1]<tv do
      i:=i+1:
    end_while:
    starti:=i:

end_proc:


err_val:=proc()
begin

    // Mean squared error computing
    errsq:=0:
    for i from starti to nops(l_data_X) do
    errsq:=errsq+(nl_data[i]-l_data[i])^2:
    end_for:
    err:=sqrt(errsq/nops(l_data_X)):
    print(Unquoted,"Mean squared error: ", err);

end_proc:


plot_val:=proc()
begin

    // Overlapped nonlinear/linear plotting
    plotpointsl:=[point(l_data_X[i][1],l_data[i]) $ hold(i)=starti..nops(l_data_X)]:
    plotpointsnl:=[point(nl_data_X[i][1],nl_data[i]) $ hold(i)=starti..nops(l_data_X)]:
    plot2d(Scaling=UnConstrained, Labeling=TRUE, Axes=Origin, AxesOrigin=[0, 0],
    Labels=["time", "state variable"], BackGround=[1.0, 1.0, 1.0],
    ForeGround=[0.0, 0.0, 0.0],
    [Mode=List, plotpointsl,Color=[Flat, [1.0, 0.0, 0.0]], Title="Linear response", PointStyle=FilledCircles],
    [Mode=List, plotpointsnl, Color=[Flat, [0.0, 0.0, 1.0]], Title="Nonlinear response", PointStyle=Circles]):

end_proc:
```

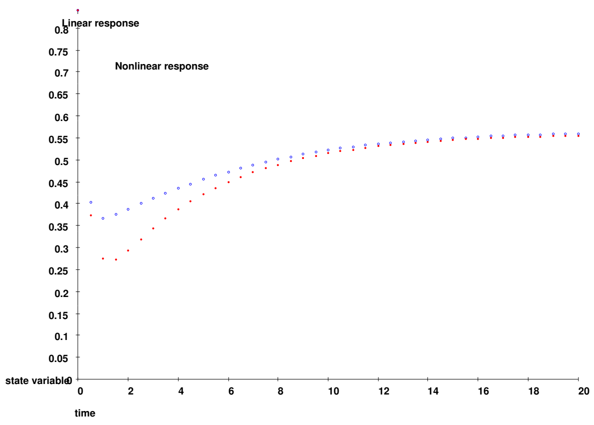
```
validation_plot:=proc()
begin

    // Mean squared error and plot printing
    repeat
        //  choose the validation starting time
        time_val():
         //  mean squared  error computing of the chosen variable
        err_val():
        print(Unquoted,"Wait please"):
        // overlapped (nonlinear/linear) plot
        plot_val():
        print(Unquoted, "a - Change the validation starting time");
        print(Unquoted, "b - Validation of other variable");
        print(Unquoted, "x - Exit");
        test_valid:=input(""):
     until test_valid<>a end_repeat:

end_proc:


valid_Xj_or_Yj:=proc()
begin

    // j-th state component or the j-th output component data
    // storing (nl/l_data[i]). Validation will be based on this data
    case test_valid

      of a do
        print(Unquoted, "State vector X = ", X_tmp);
        print();
        print(Unquoted,"type in the index corresponding to the state variable chosen ");
        print(Unquoted," for the validation");
        input("",j_plot);
        print();
        for i from 1 to nops(nl_data_X) do
          nl_data[i]:=nl_data_X[i][2][j_plot]:
        end_for:
        for i from 1 to nops(l_data_X) do
          l_data[i]:=l_data_X[i][2][j_plot]:
        end_for:
        validation_plot():
        break;

      of b do
        print(Unquoted, "Output vector Y = ", Y_tmp);
        print();
        print(Unquoted,"type in the index corresponding to the output chosen ");
        print(Unquoted," for the validation");
        input("",j_plot);
        print();
        tmp_set(F, G, X, Y, U, Unom):
        G_tmp_U_sim:=subs(G_tmp, [  U_tmp[i,1]=U_sim[i,1]    $ hold(i)=1..m ] ):
        compute_data_Yj(nl_data_X):
        for i from 1 to nops(nl_data_X) do
          nl_data[i]:=data_Yj[i]:
        end_for:
        tmp_set(F_lin, G_lin, X_lin, Y_lin, U, Unom):
        G_tmp_U_sim:=subs(G_tmp, [  U_tmp[i,1]=U_sim[i,1]    $ hold(i)=1..m ] ):
        compute_data_Yj(l_data_X):
        for i from 1 to nops(l_data_X) do
          l_data[i]:=data_Yj[i]:
```

```
        end_for:
        validation_plot():
        break;

      of x do  return():

  end_case;

end_proc:


valid_response:=proc()
begin

    // State or output options menu
    repeat
      print(Unquoted, "a - State validation");
      print(Unquoted, "b - Output validation");
      print(Unquoted, "x - Exit"):
      test_valid:=input(""):
      valid_Xj_or_Yj():
    until test_valid=x end_repeat:

end_proc:


step_size_set:=proc()
begin

  // Setting the stepsize for the "numeric::odesolve" MuPAD function
  repeat
    print(Unquoted, "enter the step size for the ODE solution:");
    step_size:=input(""):
    if step_size>t1 then    // t1 is set in pre_set_parameters()
      print(Unquoted,"the step size must be less than or equal to the final time");
    end_if:
  until step_size<=t1 end_repeat:
  print():
  print(Unquoted,"Wait, please... "):

end_proc:


validation:=proc()
begin

    // Nonlinear/linear comparison
    print():
    print(Unquoted, "Nonlinear system:");
    eqns_print(F, G):
    print(Unquoted, "Linearized system:");
    eqns_print(F_lin, G_lin):
    print();
    tmp_set(F, G, X, Y, U, Unom):
    pre_set_parameters():
    input_choice():
    step_size_set():
    compute_response_1():
    nl_data_X:=numeric::odesolve(%,Stepsize=step_size):
    tmp_set(F_lin, G_lin, X_lin, Y_lin, U, Unom):
    compute_response_1():
    l_data_X:=numeric::odesolve(%,Stepsize=step_size):
```

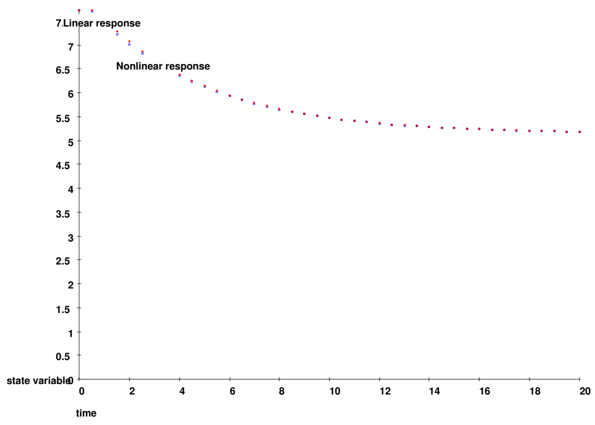
```
    valid_response():

end_proc:

time_val:=proc()
begin

    // Setting of the time at which the nonlinear and linear system
    // have to be compared
    repeat
      print(Unquoted, "enter the validation starting time: ");
      input("",tv):
      if tv>t1 then
        print(Unquoted,
        "the validation time must be less or equal to the final time");
      end_if:
    until tv<=t1 end_repeat:

    //   start time
    i:=1:
    while l_data_X[i][1]<tv do
      i:=i+1:
    end_while:
    starti:=i:

end_proc:


err_val:=proc()
begin

    // Mean squared error computing
    errsq:=0:
    for i from starti to nops(l_data_X) do
    errsq:=errsq+(nl_data[i]-l_data[i])^2:
    end_for:
    err:=sqrt(errsq/nops(l_data_X)):
    print(Unquoted,"Mean squared error: ", err);

end_proc:


plot_val:=proc()
begin

    // Overlapped nonlinear/linear plotting
    plotpointsl:=[point(l_data_X[i][1],l_data[i]) $ hold(i)=starti..nops(l_data_X)]:
    plotpointsnl:=[point(nl_data_X[i][1],nl_data[i]) $ hold(i)=starti..nops(l_data_X)]:
    plot2d(Scaling=UnConstrained, Labeling=TRUE, Axes=Origin, AxesOrigin=[0, 0],
    Labels=["time", "state variable"], BackGround=[1.0, 1.0, 1.0],
    ForeGround=[0.0, 0.0, 0.0],
    [Mode=List, plotpointsl,Color=[Flat, [1.0, 0.0, 0.0]], Title="Linear response", PointStyle=FilledCircles],
    [Mode=List, plotpointsnl, Color=[Flat, [0.0, 0.0, 1.0]], Title="Nonlinear response", PointStyle=Circles]):

end_proc:


validation_plot:=proc()
begin

    // Mean squared error and plot printing
    repeat
```

```
        //  choose the validation starting time
        time_val():
         //  mean squared  error computing of the chosen variable
        err_val():
        print(Unquoted,"Wait please"):
        // overlapped (nonlinear/linear) plot
        plot_val():
        print(Unquoted, "a - Change the validation starting time");
        print(Unquoted, "b - Validation of other variable");
        print(Unquoted, "x - Exit");
        test_valid:=input(""):
     until test_valid<>a end_repeat:

end_proc:


valid_Xj_or_Yj:=proc()
begin

    // j-th state component or the j-th output component data
    // storing (nl/l_data[i]). Validation will be based on this data
    case test_valid

      of a do
        print(Unquoted, "State vector X = ", X_tmp);
        print();
        print(Unquoted,"type in the index corresponding to the state variable chosen ");
        print(Unquoted," for the validation");
        input("",j_plot);
        print();
        for i from 1 to nops(nl_data_X) do
          nl_data[i]:=nl_data_X[i][2][j_plot]:
        end_for:
        for i from 1 to nops(l_data_X) do
          l_data[i]:=l_data_X[i][2][j_plot]:
        end_for:
        validation_plot():
        break;

      of b do
        print(Unquoted, "Output vector Y = ", Y_tmp);
        print();
        print(Unquoted,"type in the index corresponding to the output chosen ");
        print(Unquoted," for the validation");
        input("",j_plot);
        print();
        tmp_set(F, G, X, Y, U, Unom):
        G_tmp_U_sim:=subs(G_tmp, [  U_tmp[i,1]=U_sim[i,1]    $ hold(i)=1..m ] ):
        compute_data_Yj(nl_data_X):
        for i from 1 to nops(nl_data_X) do
          nl_data[i]:=data_Yj[i]:
        end_for:
        tmp_set(F_lin, G_lin, X_lin, Y_lin, U, Unom):
        G_tmp_U_sim:=subs(G_tmp, [  U_tmp[i,1]=U_sim[i,1]    $ hold(i)=1..m ] ):
        compute_data_Yj(l_data_X):
        for i from 1 to nops(l_data_X) do
          l_data[i]:=data_Yj[i]:
        end_for:
        validation_plot():
        break;

      of x do  return():
```

```
  end_case;

end_proc:


valid_response:=proc()
begin

    // State or output options menu
    repeat
      print(Unquoted, "a - State validation");
      print(Unquoted, "b - Output validation");
      print(Unquoted, "x - Exit"):
      test_valid:=input(""):
      valid_Xj_or_Yj():
    until test_valid=x end_repeat:

end_proc:


step_size_set:=proc()
begin

  // Setting the stepsize for the "numeric::odesolve" MuPAD function
  repeat
    print(Unquoted, "enter the step size for the ODE solution:");
    step_size:=input(""):
    if step_size>t1 then    // t1 is set in pre_set_parameters()
      print(Unquoted,"the step size must be less than or equal to the final time");
    end_if:
  until step_size<=t1 end_repeat:
  print():
  print(Unquoted,"Wait, please... "):

end_proc:


validation:=proc()
begin

    // Nonlinear/linear comparison
    print():
    print(Unquoted, "Nonlinear system:");
    eqns_print(F, G):
    print(Unquoted, "Linearized system:");
    eqns_print(F_lin, G_lin):
    print();
    tmp_set(F, G, X, Y, U, Unom):
    pre_set_parameters():
    input_choice():
    step_size_set():
    compute_response_1():
    nl_data_X:=numeric::odesolve(%,Stepsize=step_size):
    tmp_set(F_lin, G_lin, X_lin, Y_lin, U, Unom):
    compute_response_1():
    l_data_X:=numeric::odesolve(%,Stepsize=step_size):
    valid_response():

end_proc:
```



(a) s(t)



(b) x(t)

Figure 13.1: Validation, free response

13.3 Example Application

To substantiate our work we have made some tests comparing the response of the nonlinear system with the response of the linearized system.

1. We analyze the free response of the system with initial conditions different from $s_e = 0.56,\ x_e = 5.15$; we plot:
   s(t) and x(t) for the nonlinear system and for the linear system with initial condition equal to $s(0) = 0.84,\ x(0) = 7.725$ (figure 13.1); the mean squared error was 0.035 for s(t) and 0.021 for x(t);

2. We plot the step response of both the nonlinear system and the linearized system (figure 13.2): $d = \overline{d} + 0.1$ ($\overline{d} = 0.2$); the mean squared error was 0.051 for s(t) and 0.083 for x(t);

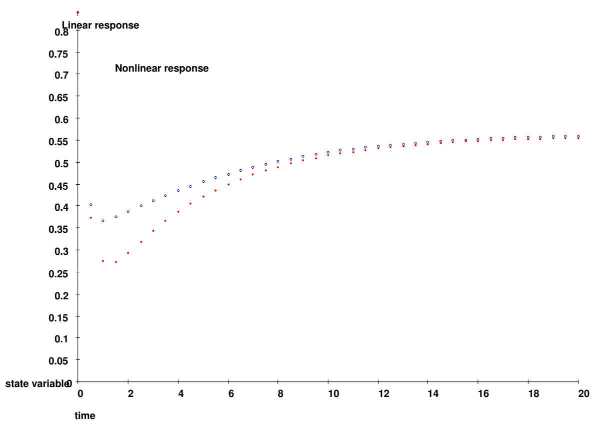
3. Sinusoidal response; we made three different tests changing the input sinusoid $d = \overline{d} + Asin(\omega t)$ ($\overline{d} = 0.2$) :

   (a) figure 13.3 A=0.02, $\omega = 1$; the mean squared error was 0.005 for s(t) and 0.004 for x(t);

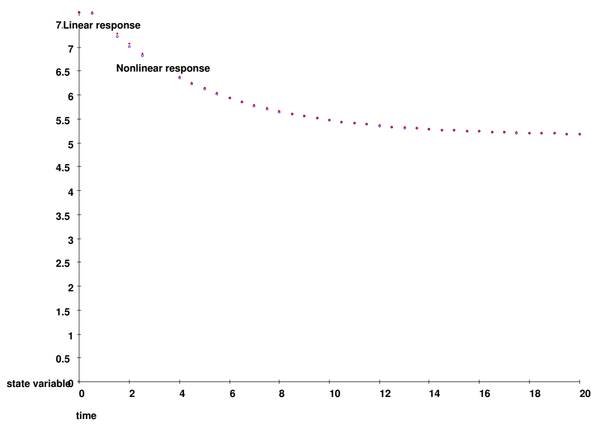
   (b) figure 13.4 A=0.1, $\omega = 1$; the mean squared error was 0.028 for s(t) and 0.019 for x(t);

   (c) figure 13.5 A=0.1, $\omega = 10$; the mean squared error was 0.007 for s(t) and 0.005 for x(t);

From these data we observe that the linear system is a good approximation to the nonlinear one; they have very similar behavior. Hence the linear system can be used to represent the nonlinear system near the equilibrium point.

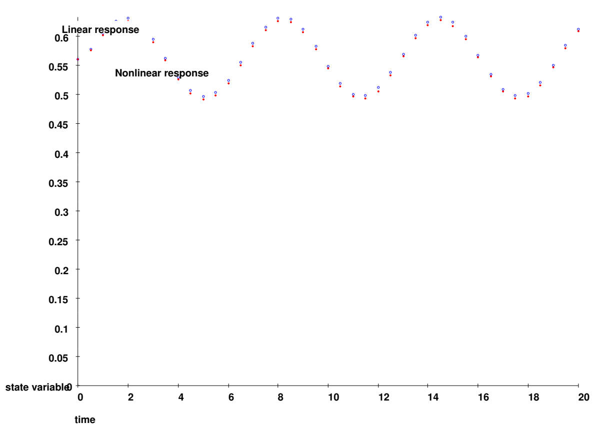


(a) s(t)

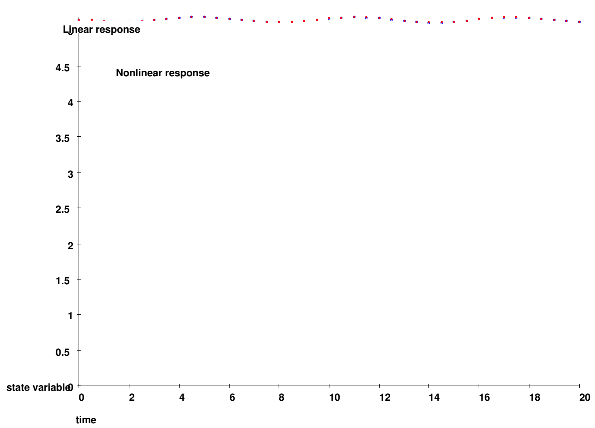


(b) x(t)

Figure 13.2: Validation, step response

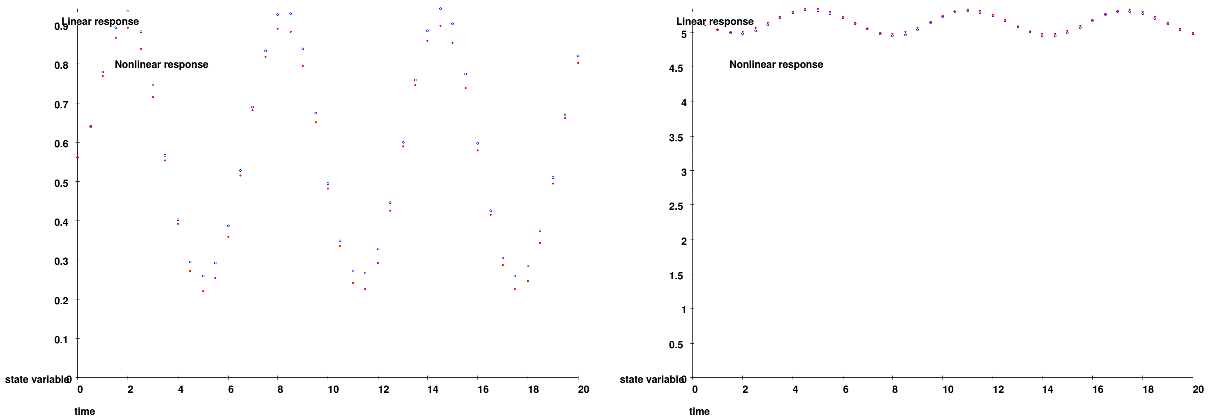


(a) s(t)



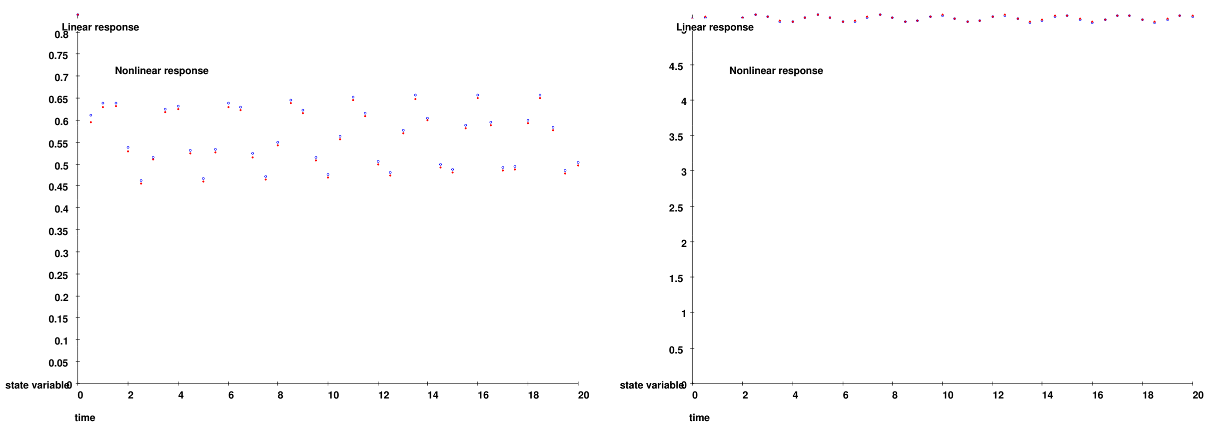
(b) x(t)

Figure 13.3: Validation, sinusoidal response , A=0.02, $\omega = 1$



(a) s(t) (b) x(t)

Figure 13.4: Validation, sinusoidal response , A=0.1, $\omega = 1$



(a) s(t) (b) x(t)

Figure 13.5: Validation, sinusoidal response , A=0.1, $\omega = 10$

14
System Time Response

14.1 Theoretical Basis

Often we need to simulate the nonlinear system behavior to check certain characteristics of the model.

14.2 MuPAD Code

The procedures used are:

- sys_time_response()
- tmp_set(); see section 16.2
- compute_response(); see section 16.4

```
sys_time_response:=proc()
begin

    // nonlinear system parameters setting and time
    // response computing
    tmp_set(F, G, X, Y, U, Unom):
    compute_response():

end_proc:
```

14.3 Example Application

We do not show any explicit example because we have already depicted the nonlinear system behavior in the comparison plots of chapter 13

15
Transfer Function Input

15.1 Theoretical Basis

If a transfer function of a linear SISO model

$$h(s) = \frac{b_r s^r + b_{r-1}s^{r-1} + \cdots + b_1 s + b_0}{a_n s^n + a_{n-1}s^{n-1} + \cdots + a_1 s + a_0}$$

(with $n \geq r$) is available it can be useful to get a state space representation of the system. This is called a “realization” of h(s). We point out that this transformation is not unique. We choose the canonical reachable realization. We will get the system:

$$\begin{cases} \frac{dx}{dt} = Ax + Bu \\ y = Cx + Du \end{cases}$$

where:

$$A = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 1 \\ -\tilde{a}_0 & -\tilde{a}_1 & \cdots & \cdots & -\tilde{a}_{n-1} \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

$$C = \begin{bmatrix} \hat{b}_0 & \hat{b}_1 & \cdots & \hat{b}_{n-1} \end{bmatrix}$$

$$D = \begin{bmatrix} \tilde{b}_n \end{bmatrix}$$

where

$$\tilde{a}_i = a_i/a_n \quad \tilde{b}_i = b_i/a_n$$

$$\hat{b}_i = \tilde{b}_i - \tilde{b}_n \tilde{a}_i$$

We observe that $[b_n \ldots b_{r+1}] = [0 \ldots 0]$. Having both the transfer function representation and the state space representation, all the above manipulations can be made.

15.2 MuPAD Code

The procedures used are:

- transfer_function_input()
- h_input()
- h_study(); see section 16.6
- IO_ISO()
- IO_ISO_transformation()
- eigenvalues(); see section 16.7
- IO_ISO_response()
- F_G_IO_compute()
- tmp_set(); see section 16.2
- compute_response(); see section 16.4

```
IO_ISO_transformation:=proc()
begin

    // IO ---> ISO transformation
    p:=1: m:=1:  // SISO system
    X:=M(n,1):   //  state vector
    Xe:=M(n,1):
    for i from 1 to n do
      X[i,1]:=x.i:
      Xe[i,1]:=0:
    end_for:
    Y:=M(p,1):
    Y[1,1]:=y:
    U:=M(m,1):
    U[1,1]:=u:
    Unom:=M(m,1):
    Unom[1,1]:=0:

    Ae:=M(n,n):
    Be:=M(n,1):
    Ce:=M(1,n):
    Die:=M(1,1):

    for i from 1 to n do
      if i=n then     // matrix Be
        Be[i,1]:=1:
      else Be[i,1]:=0:
      end_if:
      Ce[1,i]:=num_arr[i-1]/den_arr[n]
         -num_arr[n]*den_arr[i-1]/(den_arr[n]*den_arr[n]):  // matrix Ce
      for j from 1 to n do
        if i=n then                // matrix Ae
          Ae[i,j]:=-den_arr[j-1]/den_arr[n]:
        elif j=i+1 then
          Ae[i,j]:=1:
        else Ae[i,j]:=0:
        end_if:
```

```
      end_for:
    end_for:

    Die[1,1]:=num_arr[n]/den_arr[n]:     // matrix Die

end_proc:


F_G_IO_compute:=proc()
begin

    // Starting from matrix representation, write the corresponding
    // equations.
    // Xe=0, Unom=0
    F:=Ae*X+Be*U:
    G:=Ce*X+Die*U:

end_proc:


IO_ISO_response:=proc()
begin

    //
    tmp_set(F, G, X, Y, U, Unom):
    compute_response():

end_proc:


IO_ISO:=proc()
local test;
begin

    // IO ---> I/S/O transformation and options menu
    IO_ISO_transformation():
    repeat
      print(Unquoted, "A = ", Ae);
      print(Unquoted, "B = ", Be);
      print(Unquoted, "C = ", Ce);
      print(Unquoted, "D = ", Die);
      print();
      F_G_IO_compute():
      print(Unquoted, "Right hand side of the state equations:");
      print(F);
      print(Unquoted, "Right hand side of the output equation:");
      print(G);
      print();
      print(Unquoted, "a - Eigenvalues");
      print(Unquoted, "b - System time response");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do eigenvalues(): break:
        of b do IO_ISO_response(): break:
      end_case:
    until test=x end_repeat:

end_proc:
```

```
h_input:=proc()
local test;
begin

    // Transfer function input and causality check
    print(Unquoted,"h(s)=(b[r]s^r+...b[1]s+b[0])/(a[n]s^n+...+a[1]s+a[0])");
    repeat
      r:=input("r:");
      print();
      n:=input("n:");
      if r>n then
         print(Unquoted,"r must be less or equal to n");
      end_if:
    until r<=n end_repeat:

    b1:= array(0..r):
    den_arr:= array(0..n):
    num_arr:=array(0..n):
    repeat
      for i from 0 to r do
        print(Unquoted, "b[".i."]"):
        b1[i]:=input("");
      end_for;
      print(b1);
      input("Are these data right ? (y/n): ", test):
    until test=y end_repeat;

    // num_arr: array of n+1 elements: the first r+1 are equals to b1
    //the last n-r are  equal to zero

    for i from 0 to r do
      num_arr[i]:=b1[i]:
    end_for:
    for i from r+1 to n do
      num_arr[i]:=0:
    end_for:

    repeat
      for i from 0 to n do
        print(Unquoted, "a[".i."]"):
        den_arr[i]:=input("");
      end_for;
      print(den_arr);
      input("Are these data right ? (y/n): ", test):
      if den_arr[n]=0 then
        print(Unquoted,"a[n] must be different from zero "):
        test:=n:
      end_if:
    until test=y end_repeat:

    unassign(s):
    numer_h:=0:
    for i from 0 to r do
      numer_h:=numer_h+num_arr[i]*s^i
    end_for:
    den_h:=0:
    for i from 0 to n do
      den_h:=den_h+den_arr[i]*s^i
    end_for:
    h(s):=numer_h/den_h:

end_proc:
```

```
transfer_function_input:=proc()
local test;
begin

    // Transfer function input and options menu
    h_input():
    repeat
      print();
      print(Unquoted, "h(s)=", h(s));
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Transfer function study/Control design");
      print(Unquoted, "b - I/O ---> I/S/O");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do h_study(h(s)): break:
        of b do IO_ISO(): break:
      end_case:
    until test=x end_repeat:

end_proc:
```

15.3 Example Application

We consider the following transfer function:

$$h(s) = \frac{6.01s + 1.18}{0.64s^2 + 1.8s + 0.36}$$

The gain h(0) is 3.28 and we can rewrite h(s) in the zeros–poles form:

$$h(s) = 9.38\frac{s + 0.2}{(s + 0.22)(s + 2.6)}$$

The canonical reachable realization of h(s) is:

$$A = \begin{bmatrix} 0 & 1 \\ -0.5625 & -2.8125 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$

$$C = \begin{bmatrix} 1.84375 & 9.3906 \end{bmatrix}, \quad D = \begin{bmatrix} 0 \end{bmatrix}$$

From this state representation we have plot the following:

1. figure 15.1: y(t) with u(t)= $0.2 + 0.02sin(t)$
2. figure 15.2: y(t) with u(t)= $0.2 + 0.1sin(t)$
3. figure 15.3: y(t) with u(t)= 0.3

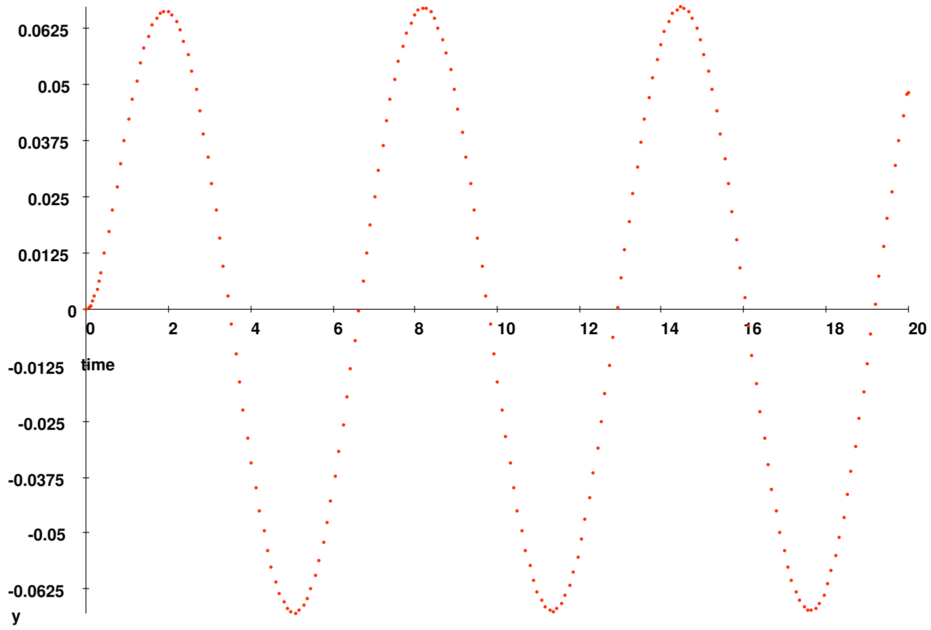


Figure 15.1: y(t) plot, u(t)= $0.2 + 0.02sin(t)$

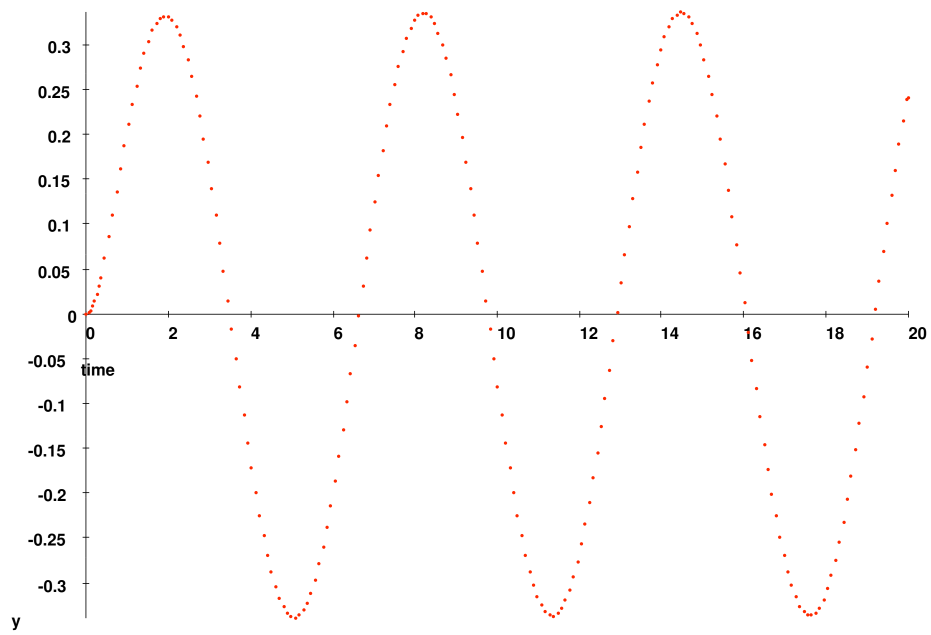


Figure 15.2: y(t) plot, u(t)= $0.2 + 0.1sin(t)$

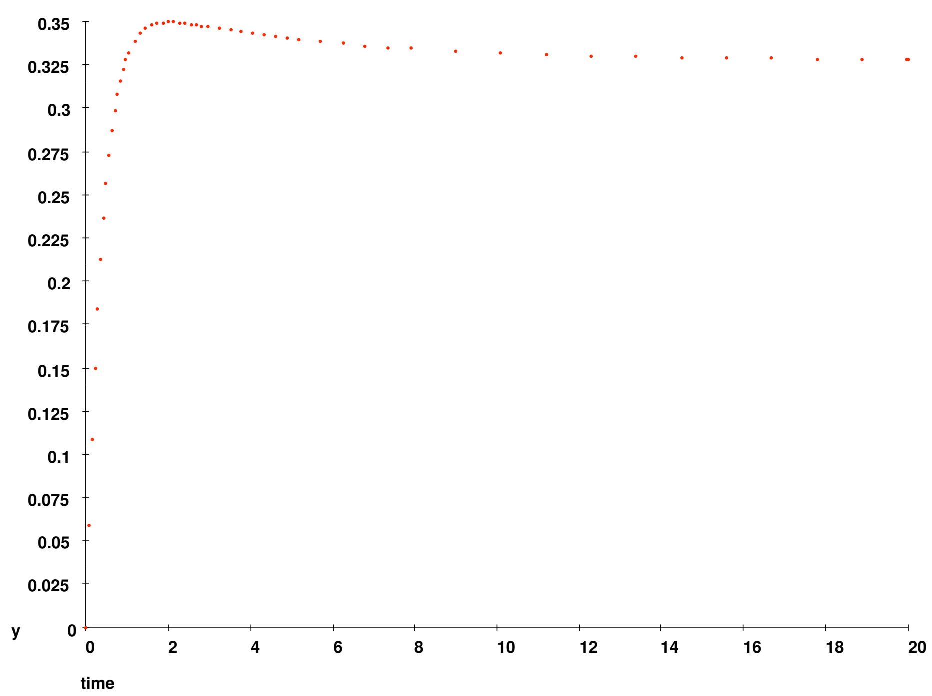


Figure 15.3: y(t) plot, u(t)= 0.3

16
Common Procedures

This chapter contains the code of the procedures used by several routines.

16.1 roots_finding procedure

```
roots_finding:=proc(pol)
  local n,data,data_tmp,q,i,j,k;
begin

     // get the list of all the roots of a polynomial in s: if there are
     // multiple roots they will appear a number of times equal to their
     // multiplicity.
     // NB: pol must be a polynomial in the variable s
     n := degree(pol);
     data:=numeric::bairstow(pol);
     if data=FAIL then
       return(FAIL)
     end_if:
     if nops(data) < n then
       k := 0;
       data_tmp := array(1..n);
       for i from 1 to nops(data) do
         q:=pol;
         repeat
           q_r:=divide(q,(s-op(data,i)));
           q:=op(q_r,1);
           r:=float(op(q_r,2));
           if r=0.0 then
             k:=k+1;
             data_tmp[k]:=op(data,i);
           end_if:
         until r<>0.0 end_repeat:
       end_for;
       data:=data_tmp;
     end_if;
     data;

end_proc:
```

16.2 tmp_set procedure

```
tmp_set:=proc(F_formal, G_formal, X_formal, Y_formal, U_formal, Unom_formal)
begin

     // F_tmp, G_tmp,etc.. updating
     F_tmp:=F_formal:
     G_tmp:=G_formal:
     X_tmp:=X_formal:
     Y_tmp:=Y_formal:
     U_tmp:=U_formal:
     Unom_tmp:=Unom_formal:

end_proc:
```

16.3 real_immaginary, amplitude, phase procedure

```
real_imaginary:=proc(h_formal)
  local hw,recthw;
begin

    // Real and imaginary part a transfer function h
    // frequency response hw(w)= h(s=I*w)
    print():
    print(Unquoted, "Wait, please... "):
    hw(w):=subs(h_formal,s=I*w):

    // rectangular form of h(jw): rectform is useful also to explain
    // that w is not complex
    recthw(w):=rectform(hw(w),{w}):

    // real and imaginary part useful also to Nyquist plot (global variables)
    rehw(w):=expand(Re(recthw(w))):
    imhw(w):=expand(Im(recthw(w))):

end_proc:


amplitude:=proc(h_formal)
  local rehw2,inhw2,amplhw2,amplhw;
begin

    // Amplitude of a transfer function
    real_imaginary(h_formal);
    // rehw and imhw square
    rehw2(w):=expand(rehw(w)^2):
    imhw2(w):=expand(imhw(w)^2):

    // squared amplitude and phase useful to Bode plot
    amplhw2(w):=expand(rehw2(w)+imhw2(w)):

    // functions must be declared f(w) (depending on the parameter
    // w before plotting)
    amplhw(w):=sqrt(amplhw2(w));

end_proc:
```

```
phase:=proc(h_formal)
local hnum,hden,n_hnum,n_hden,z_hnum,z_hden,
      facthnum,facthnumrect,facthnumre,facthnumim,phasehnum,
      facthden,facthdenrect,facthdenre,facthdenim,phasehden,
      k,p_tmp,hpz0,phase;
begin

    // Phase: we utilize the phase property:
    // phase(a*b/c)=phase(a)+phase(b)-phase(c)
    // phase will be in degrees

    // phase of hw(w)
    // numerator, denominator, degrees
    hnum(s):=numer(h_formal):
    hden(s):=denom(h_formal):
    n_hnum:=degree(hnum(s),s):
    n_hden:=degree(hden(s),s):

    // numerator zeros and factorized form
    if n_hnum>0 then
      z_hnum:=roots_finding(hnum(s)):   // numerator zeros
      if z_hnum=FAIL then return(FAIL) end_if:
      facthnum(s):=(s-op(z_hnum,1)):
      for i from 2 to n_hnum do
        facthnum(s):=facthnum(s)*(s-op(z_hnum,i)):
      end_for;
    else facthnum(s):=hnum(s);
    end_if;

    // denominator zeros and factorized form
    if n_hden>0 then
      z_hden:=roots_finding(hden(s)):   // denominator zeros
      if z_hden=FAIL then return(FAIL) end_if:
      facthden(s):=(s-op(z_hden,1)):
      for i from 2 to n_hden do
        facthden(s):=facthden(s)*(s-op(z_hden,i));
      end_for;
    else facthden(s):=hden(s);
    end_if;

    // in each factor we substitute s=I*w
    for i from 1 to n_hnum do
      facthnum(w,i):=subs(s-op(z_hnum,i),s=I*w):
      facthnumrect(w,i):=rectform(facthnum(w,i),{w}):
      facthnumre(i):=expand(Re(facthnumrect(w,i))):
      facthnumim(i):=expand(Im(facthnumrect(w,i))):
    end_for:

    // hw(w) numerator phase is equal to the sum of the phase of each
    // term facthnum(i)
    phasehnum(w):=0:
    for i from 1 to n_hnum do
      phasehnum(w):=phasehnum(w)+atan(facthnumre(i),facthnumim(i))
    end_for:

    // like numerator
    for i from 1 to n_hden do
      facthden(w,i):=subs(s-op(z_hden,i),s=I*w):
      facthdenrect(w,i):=rectform(facthden(w,i),{w}):
      facthdenre(i):=expand(Re(facthdenrect(w,i))):
      facthdenim(i):=expand(Im(facthdenrect(w,i))):
```

```
    end_for:

    // like numerator
    phasehden(w):=0:
    for i from 1 to n_hden do
      phasehden(w):=phasehden(w)+atan(facthdenre(i),facthdenim(i))
    end_for:

    // gain:
    hpz(s):=facthnum(s)/facthden(s);
    // h(0) or h(1) or h(2) or ...
    i:=-1:
    repeat
      i:=i+1:
    until (float(op(divide(hden(s),s-i),2))<>0.0
      and float(op(divide(hnum(s),s-i),2))<>0.0)
    end_repeat:
    // the gain hpz(0) (hpz(1)) must be the same of h(0) (h(1))
    h0:=subs(h(s),s=i):
    hpz0:=Re(subs(hpz(s),s=i));
    k:=h0/hpz0;
    k:=float(k);

    // hw(w) phase is equal to the difference between the numerator phase and
    // the denominator phase
    if Re(k)>=0 then
      ph0:=0:
    else
      ph0:=-PI:
    end_if:
    phase(w):=ph0+phasehnum(w)-phasehden(w):
    phase(w):=phase(w)*180/PI;

end_proc:
```

16.4 compute_response procedure

```
compute_response:=proc()
begin

    //
    pre_set_parameters():
    input_choice():
    print():
    print(Unquoted, "Wait, please... "):
    compute_response_1():
    data_X:=numeric::odesolve(%):
    response_plot():

end_proc:
```

16.4.1 pre_set_parameters procedure

```
pre_set_parameters:=proc()
local test;
```

```
begin

    // Initial values, starting and stopping time setting
    // for ODE solution
    repeat
      print(Unquoted, "enter data for the simulation");
      print();
      for i from 1 to n do
        print(Unquoted,"enter initial condition for the variable ".expr2text(X_tmp[i,1]));
        Y0[i]:=input("");
      end_for;
      print(Unquoted,"enter simulation initial time: ");
      t0:=input(""):
      print(Unquoted,"enter simulation final time: ");
      t1:=input(""):
      input("Are these data right ? (y/n): ", test):
    until test=y end_repeat:

end_proc:
```

16.4.2 input_choice procedure

```
input_choice:=proc()
local test;
begin

    // Input function setting to add to its nominal value
    U_sim:=M(m,1):
    U_in:=M(m,1):
    repeat
      for i from 1 to m do
        print(Unquoted, "Input variable:  " .U_tmp[i,1]);
        print(Unquoted, "Nominal value:  " .expr2text(Unom_tmp[i,1])):
        print(Unquoted,
          "Enter the function in the variable t, f(t), to add to "
          .expr2text(Unom_tmp[i,1])):
        U_in[i,1]:=input(""):
      end_for:
      print(U_in);
      input("Are these data right ? (y/n): ", test):
    until test=y end_repeat:
    U_sim:=Unom_tmp+U_in:

end_proc:
```

16.4.3 compute_response_1 procedure

```
compute_response_1:=proc()
begin

   // F_tmp evaluation in the U_sim input and parameters setting
   // for ODE solving
   F_tmp:=subs(F_tmp, [  U_tmp[i,1]=U_sim[i,1]    $ hold(i)=1..m ] ):
   set_parameters():

end_proc:
```

set_parameters procedure

```
set_parameters:=proc()
local y,y0,F_sim,FF,i,T0,T1,
      timeinterval,vectorfield,initialconditions,myoptions;
begin

   // This procedure substitutes the X_tmp[i] variables (user
   // defined labels) in the F_tmp  vector with internal variables
   // y[1]..y[n]; the result is stored in the F_sim list. This is
   // very important since the "numeric::odesolve" MuPAD procedure
   // needs the system of equations expressed as a function of a
   // known vector (y)
   F_sim:=[0$n];   # initialize list for equations #
   for i from 1 to n do
       F_sim[i]:=F_tmp[i,1];
   end_for;
   F_sim:= subs(F_sim, [ X_tmp[i,1]=y[i] $ hold(i)=1..n ] );
   y0:=[0$n];   # initialize list for initial data #
   for i from 1 to n do
     y0[i]:=Y0[i]:
   end_for;
   T0:=t0:
   T1:=t1:
   timeinterval:= T0..T1:
   vectorfield := subs( proc(t,y) begin FF end_proc, FF=F_sim):
   initialconditions:= y0;
   myoptions:= Alldata=1;
   // The following parameters are to be passed to odesolve
   timeinterval,vectorfield,initialconditions,myoptions;

end_proc:
```

16.4.4 response_plot procedure

```
response_plot:=proc()
begin

    // State or output plot options menu
    repeat
      print();
      print(Unquoted, "Options");
      print(Unquoted, "a - State variable plot");
      print(Unquoted, "b - Output variable plot");
      print(Unquoted, "x - exit");
      print();

      input("", test1):
      plot_Xj_or_Yj():
    until test1=x end_repeat;


end_proc:
```

plot_Xj_or_Yj procedure

```
plot_Xj_or_Yj:=proc()
begin

  // plot of the j-th state component or the j-th output component
  case test1

    of a do
      print(Unquoted, "State vector X = ", X_tmp);
      print();
      input("type in the index corresponding to the state variable to be plotted:
            ", j_plot):
      print();
      print(Unquoted, "State component to be plotted: ");
      print(X_tmp[j_plot, 1]);
      label_plot:=expr2text(X_tmp[j_plot, 1]):
      for i from 1 to nops(data_X) do
        data_plot[i]:=data_X[i][2][j_plot]:
      end_for:
      break;

    of b do
      print(Unquoted, "Output vector Y = ", Y_tmp);
      print();
      input("type in the index corresponding to the output variable to be plotted:
            ", j_plot);
      print();
      print(Unquoted, "Output component to be plotted: ");
      print(Y_tmp[j_plot, 1]);
      label_plot:=expr2text(Y_tmp[j_plot, 1]):
      G_tmp_U_sim:=subs(G_tmp, [  U_tmp[i,1]=U_sim[i,1]    $ hold(i)=1..m ] ):
      compute_data_Yj(data_X):
      for i from 1 to nops(data_X) do
        data_plot[i]:=data_Yj[i]:
      end_for:
      break;

    otherwise return():

  end_case;

  plotpoints:=[point(data_X[i][1], data_plot[i]) $ hold(i)=1..nops(data_X)]:

  plot2d(Scaling=UnConstrained, Labeling=TRUE, Axes=Origin, AxesOrigin=[0, 0],
  Labels=["time", label_plot], BackGround=[1.0, 1.0, 1.0],
  ForeGround=[0.0, 0.0, 0.0],
  [Mode=List, plotpoints, PointStyle=FilledCircles]):

end_proc:
```

compute_data_Yj procedure

```
compute_data_Yj:=proc(data_X_formal)
begin
```

```
    // Output points computing based on the state points
    // (data_X_formal), obtained via "numeric::odesolve"
    for i from 1 to nops(data_X_formal) do

      // The following row of code is useful in case of input
      // t dependence:
      //  we must evaluate the output equations G_tmp_U_sim
      // (they could depend on the input vector) at the time
      // instants returned  by "odesolve"

      G_tmp_U_sim_eval[j_plot, 1]:=
         float(subs(G_tmp_U_sim[j_plot, 1], t=data_X_formal[i][1])):

      data_Yj[i]:=
         subs(G_tmp_U_sim_eval[j_plot, 1],
         [ X_tmp[k,1]=data_X_formal[i][2][k] $ hold(k)=1..n ]   ):
    end_for:

end_proc:
```

16.5 eqns_print procedure

```
eqns_print:=proc(F_formal, G_formal)
begin

    // system of equations printing
    print();
    print(Unquoted, "Right hand side of the state equation:");
    print(F_formal);
    print(Unquoted, "Right hand side of the output equation:");
    print(G_formal);

end_proc:
```

16.6 h_study procedure

```
h_study:=proc(h_formal)
local test;
begin

    // Transfer function study options menu
    repeat
      print();
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Pole-zero representation/Gain");
      print(Unquoted, "b - Bode, Nyquist, Nichols plots");
      print(Unquoted, "c - Control design");
      print(Unquoted, "x - Exit");
      print();
      input("",test);
      case test
        of a do pole_zero(h_formal): break:
        of b do plots(h_formal): break:
        of c do control_menu(): break:
```

```
  end_case:
until test=x end_repeat:

end_proc:
```

16.6.1 pole_zero procedure

```
pole_zero:=proc(h_formal)
local i;
begin

    // Pole-zero recover of a transfer function
    // h(s) numerator, denominator and their degrees
    hnum(s):=numer(h_formal):
    hden(s):=denom(h_formal):
    n_hnum:=degree(hnum(s),s):
    n_hden:=degree(hden(s),s):

    // numerator in factorized form
    if n_hnum>0 then
      z_hnum:=roots_finding(hnum(s)):   // numerator zeros
      if z_hnum=FAIL then return(FAIL) end_if:
      facthnum(s):=(s-op(z_hnum,1)):
      for i from 2 to n_hnum do
        facthnum(s):=facthnum(s)*(s-op(z_hnum,i)):
      end_for;
    else facthnum(s):=hnum(s);
    end_if;

    // denominator in factorized form
    if n_hden>0 then
      z_hden:=roots_finding(hden(s)):   // denominator zeros
      if z_hden=FAIL then return(FAIL) end_if:
      facthden(s):=(s-op(z_hden,1)):
      for i from 2 to n_hden do
        facthden(s):=facthden(s)*(s-op(z_hden,i));
      end_for;
    else facthden(s):=hden(s);
    end_if;

    // h_formal in pole-zero form: hpz(s)
    i:=-1:
    repeat
      i:=i+1:
    until (float(op(divide(hden(s),s-i),2))<>0.0
      and float(op(divide(hnum(s),s-i),2))<>0.0)
    end_repeat:
    hpz(s):=facthnum(s)/facthden(s);   // wrong gain !
    // the gain hpz(0) must be the same as h_formal(0)
    h0:=subs(h_formal,s=i):
    if i=0 then
      print(Unquoted,
     " the gain (transfer function evalued in s=0) is:" ,float(h0));
    end_if:
    hpz0:=Re(subs(hpz(s),s=i));
    k:=h0/hpz0;
    k:=float(k);
    hpz(s):=k*hpz(s):    // this is the right hpz(s)
    print(Unquoted,"The pole-zero representation is:");
    print(float(hpz(s))):
```

```
end_proc:
```

16.6.2 plots procedure

```
common_plot:=proc(h_formal)
begin

    // amplitude in decibels and phase of h_formal(Iw)
    // global variables
    amplhwdb(w):=20*ln(amplitude(h_formal))/ln(10):
    phasehw(w):=phase(h_formal);

end_proc:


bode_plot:=proc(h_formal)
begin

    // Bode amplitude and phase plot
    common_plot(h_formal):

    // Bode plot: Amplitude
    plot2d(Scaling=UnConstrained, Labeling=TRUE, Axes=Origin,
    AxesOrigin=[0, 0],   Labels=["frequency ln(w)", "|h(jw)|dB"],
    BackGround=[1.0, 1.0, 1.0],
    ForeGround=[0.0, 0.0, 0.0],
    [Mode=Curve, [ln(w),amplhwdb(w)], w=  [0.01, 100],
    Grid=[50], Smoothness=[20]]);

    print(Unquoted, "Press ’c’ to continue");
    input("");

    // Bode plot: Phase
    plot2d(Scaling=UnConstrained, Labeling=TRUE, Axes=Origin,
    AxesOrigin=[0, 0],   Labels=["frequency ln(w)", "Phase"],
    BackGround=[1.0, 1.0, 1.0],
    ForeGround=[0.0, 0.0, 0.0],
    [Mode=Curve, [ln(w), phasehw(w)], w=[0.01, 100],
    Grid=[50], Smoothness=[20]]);

end_proc:


nyquist_plot:=proc(h_formal)
local w0,w1;
begin

    // Nyquist plot
    real_imaginary(h_formal):

    print(Unquoted, "Enter the initial value for w (w>0)");
    w0:=input(""):
    print(Unquoted, "Enter the final value for w");
    w1:=input(""):
    plot2d(Scaling=UnConstrained, Labeling=TRUE, Axes=Origin,
    AxesOrigin=[0, 0],   Labels=["Re[h(jw)]", "Imh[(jw)]"],
    BackGround=[1.0, 1.0, 1.0],
    ForeGround=[0.0, 0.0, 0.0],
```

```
    [Mode=Curve, [float(rehw(w)), float(imhw(w))], w=[w0,w1],
    Grid=[50], Smoothness=[20]]);

end_proc:


nichols_plot:=proc(h_formal)
begin

    // Nichols plot
    common_plot(h_formal):

    plot2d(Scaling=UnConstrained, Labeling=TRUE, Axes=Origin,
    AxesOrigin=[0, 0],   Labels=["Phase", "|h(jw)|dB"],
    BackGround=[1.0, 1.0, 1.0],
    ForeGround=[0.0, 0.0, 0.0],
    [Mode=Curve, [phasehw(w),amplhwdb(w)], w=[0.01,   100],
    Grid=[50], Smoothness=[20]]);

end_proc:

plots:=proc(h_formal)
local test;
begin

    // Bode, Nyquist, Nichols plots  menu
    repeat
      print(Unquoted, "Options: ");
      print(Unquoted, "a - Bode plot");
      print(Unquoted, "b - Nyquist plot");
      print(Unquoted, "c - Nichols plot");
      print(Unquoted, "x - Exit");
      print();
      input("", test);

      case test
        of a do bode_plot(h_formal): break;
        of b do nyquist_plot(h_formal): break;
        of c do nichols_plot(h_formal): break;
      end_case;
    until test=x end_repeat;

end_proc:
```

16.7 eigenvalues procedure

```
eigenvalues:=proc()
begin

    // Stability information based on eigenvalues
    print();
    print(Unquoted, "Please, wait...");
    eigenvls:=linalg::eigenValues(Ae,All):
    print(Unquoted, "The eigenvalues of the linearized model are: " );
    print(float(eigenvls));

end_proc:
```

Part III

AERATION TANK

17
Introduction

This part describes the application of PSYCO to analyze and control a simple aerated activated sludge reactor. Figure 17.1 shows a biological plant without recycle of sludge, which aims to remove the substrate contained in the influent water by means of biological reactions. In the reactor, which is completely mixed, heterotrophic biomass is present. It biodegrades the substrate thus removed. This type of plant is not built in practice, but the functioning of other plant types is quite similar to this one.

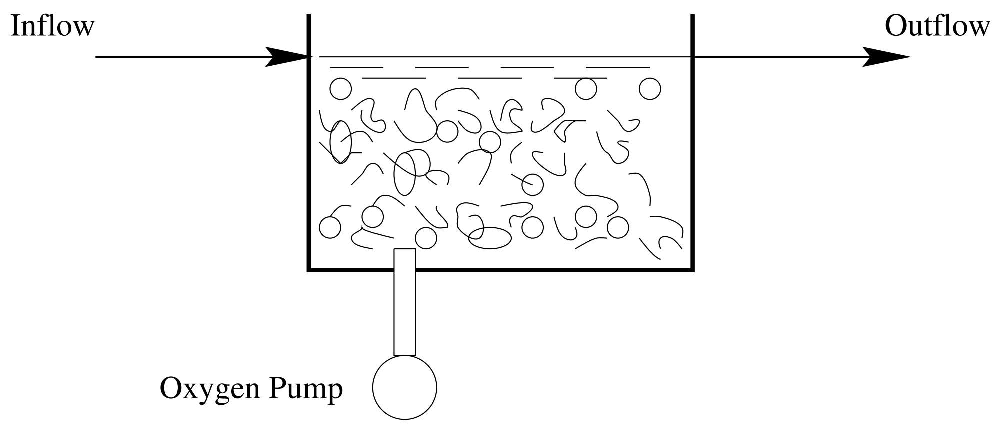


Figure 17.1: Aeration tank

The plant model considered is a simplified version of the IAWQ1 (section 4.3). The processes incorporated in the model are: growth of heterotrophic biomass and decay of heterotrophic biomass. The following three variables are crucial to the model: the oxygen concentration, the substrate concentration and the biomass concentration. In the context of modeling the effluent quality, the following model can be considered:

$$\begin{cases} \frac{dS_o}{dt} = Kla(S_o^* - S_o) + \frac{Q}{V}(S_o^{in} - S_o) - r_o \\ \frac{dS_s}{dt} = \frac{Q}{V}(S_s^{in} - S_s) - \frac{r_{growth}}{Y_H} \\ \frac{dX_{BH}}{dt} = \frac{Q}{V}(X_{BH}^{in} - X_{BH}) + r_{growth} - r_{decay} \\ r_o = \frac{1-Y_H}{Y_H} r_{growth} + (1 - f_P) r_{decay} \\ r_{growth} = \mu_{max} \frac{S_o}{K_{so}+S_o} \frac{S_s}{K_{ss}+S_s} X_{BH} \\ r_{decay} = b_H X_{BH} \end{cases}$$

where

- S_o, S_s and X_{BH} denote, respectively, the oxygen, substrate and heterotrophic biomass concentrations in the tank
- S_o^{in}, S_s^{in} and X_{BH}^{in} are same concentrations in the influent
- $\frac{Q}{V}$ is the flow rate
- Kla, S_o^* are the oxygen transfer coefficient and saturation coefficient
- Y_H is the yield for heterotrophic biomass
- f_P denotes the fraction of biomass converted to inert matter
- μ_{max} is the maximum specific growth rate for heterotrophic biomass
- K_{so}, K_{ss} are the oxygen and substrate half-saturation coefficient for heterotrophic biomass
- b_H denotes the decay coefficient for heterotrophic biomass
- r_o is the oxygen uptake rate (OUR)
- r_{growth}, r_{decay} are the aerobic growth and decay of heterotrophs

In each differential equation there is a transport term $\frac{Q}{V}(C^{in} - C)\ \ (C : concentration)$. For the S_o there is also an aeration factor $(Kla(S_o^* - S_o))$ due to the oxygen pumped into the tank. The other terms are due to the biological processes.
The parameters values used in the study reported here are listed in Table 17.1 .

Symbol	Unit	Value at $20°C$
Y_H	g cell COD formed	0.67
K_{so}	mgO_2/l	0.20
K_{ss}	$mgCOD/l$	20
b_H	day^{-1}	0.62
μ_{max}	day^{-1}	6
S_o^*	mg/l	9.1

Table 17.1: Typical parameters values at neutral pH

Furthermore it is assumed that in the influent there are no oxygen nor biomass, *i.e.,* S_o^{in} and X_{BH}^{in} are equal to zero. The flow rate (i.e., Q/V) has a nominal value of 2 day^{-1}. S_s^{in} is fixed to 1000 mg/l. Kla has a range of $[48, 144]$ day^{-1} and the nominal value is 50 day^{-1}.
The purpose of the biological plant is to remove the substrate contained in the influent water by means of biological reactions. To attain this aim we need a good aeration in the tank so to let the biomass grow and consequently to decrease the substrate. Figure 17.2 shows the control scheme applied to the plant. The controller acts on the oxygen pump, *i.e.,* on the oxygen transfer coefficient (Kla). The goal is to fix the oxygen concentration equal to 2 mg/l ($S_o \rightarrow 2mg/l$).

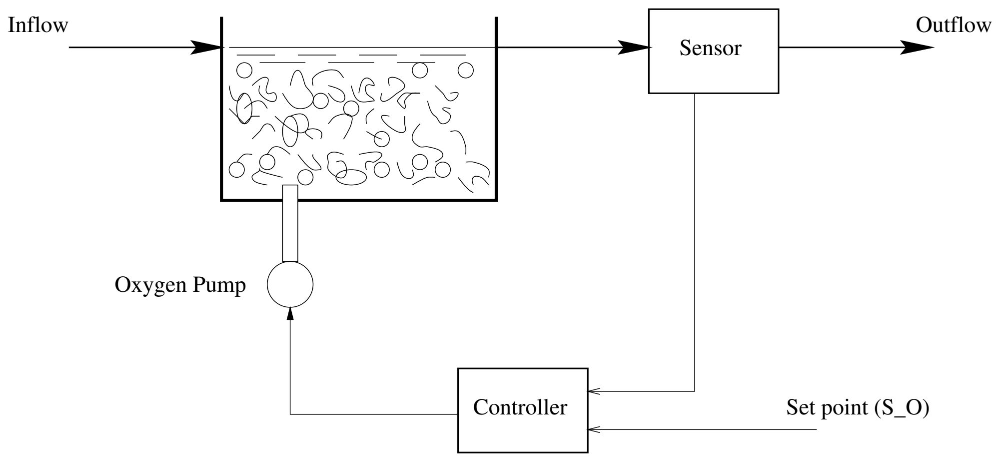


Figure 17.2: Control scheme of an aeration tank

18

Aeration Tank: Analysis

18.1 State Space Representation

The biological plant model

$$\begin{cases} \frac{dS_o}{dt} = Kla(S_o^* - S_o) + \frac{Q}{V}(S_o^{in} - S_o) - r_o \\ \frac{dS_s}{dt} = \frac{Q}{V}(S_s^{in} - S_s) - \frac{r_{growth}}{Y_H} \\ \frac{dX_{BH}}{dt} = \frac{Q}{V}(X_{BH}^{in} - X_{BH}) + r_{growth} - r_{decay} \\ r_o = \frac{1-Y_H}{Y_H} r_{growth} + (1 - f_P) r_{decay} \\ r_{growth} = \mu_{max} \frac{S_o}{K_{so}+S_o} \frac{S_s}{K_{ss}+S_s} X_{BH} \\ r_{decay} = b_H X_{BH} \end{cases}$$

is a state space representation of a nonlinear time variant model with three state variables: S_o, S_s, X_{BH}. In this analysis it is assumed that the flow rate and the concentration of the substrate in the influent are constant and equal to their nominal values ($Q/V = 2day^{-1}$, $S_s^{in} = 1000mg/l$). The oxygen transfer coefficient, Kla, is the manipulated variable (i.e. the input of the system) and the oxygen concentration in the effluent, S_o, is the observable variable (i.e. the output of the system). Hence, with the parameters values from Table 17.1, the system becomes:

$$\begin{cases} \frac{dS_o}{dt} = Kla(9.1 - S_o) - 2S_o - 0.5704X_{BH} - \frac{2.95522388 S_o S_s X_{BH}}{(S_s+20)(S_o+0.2)} \\ \frac{dS_s}{dt} = 2000 - 2S_s - \frac{8.95522388 S_o S_s X_{BH}}{(S_s+20)(S_o+0.2)} \\ \frac{dX_{BH}}{dt} = -2.62X_{BH} + \frac{6 S_o S_s X_{BH}}{(S_s+20)(S_o+0.2)} \\ y = S_o \end{cases}$$

When PSYCO is run, a menu appears:

- State variable system input
- Transfer function input
- quit

Choosing the first option, the right hand side of the above system is the input required by PSYCO. Also Kla nominal value ($50day^{-1}$) is necessary. Then, PSYCO gives the choice:

- Linearization
- System time response
- Exit

18.2 System Time Response

We want analyze the system behavior with:

- $Kla = 50day^{-1}$ (the nominal value)
- state initial conditions:
  - $S_o = 0.2mg/l$
  - $S_s = 400mg/l$
  - $X_{BH} = 300mg/l$

With PSYCO, setting these values and the simulation time parameters, we can obtain the three plots: S_o, S_s, X_{BH} free response (figures 18.1, 18.2, 18.3).

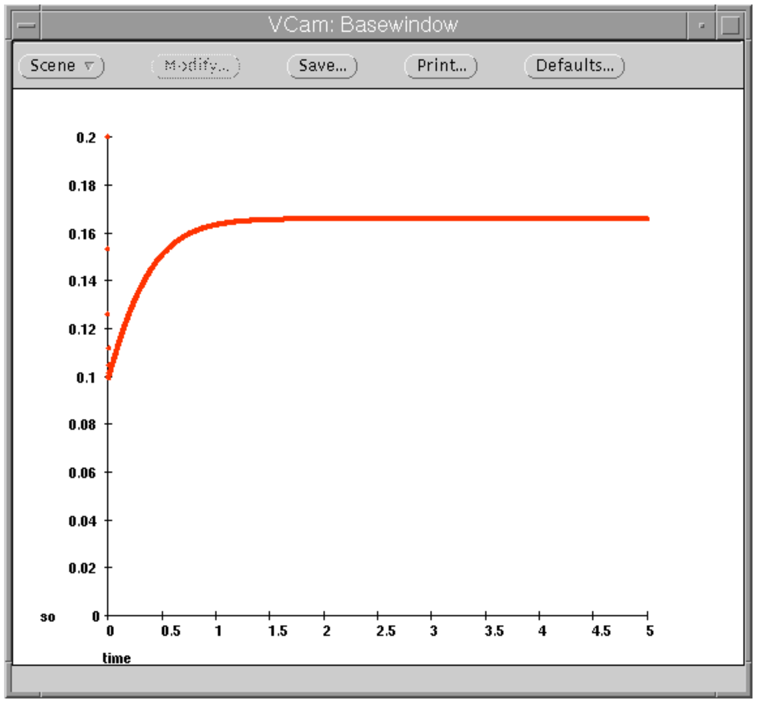


Figure 18.1: S_o, free response

We obtain the same plots using WEST++. From the modeling environment in the HGE window we have to draw the scheme of the system (figure 18.4).
For each icon multiple models had to be built. Appendix A) shows the MSL-USER library built for this example.
Then from the experimentation environment we can set the parameters values, the initial conditions, the simulation time, choose the integrator and let start the simulation; figure 18.5 shows the behavior obtained.

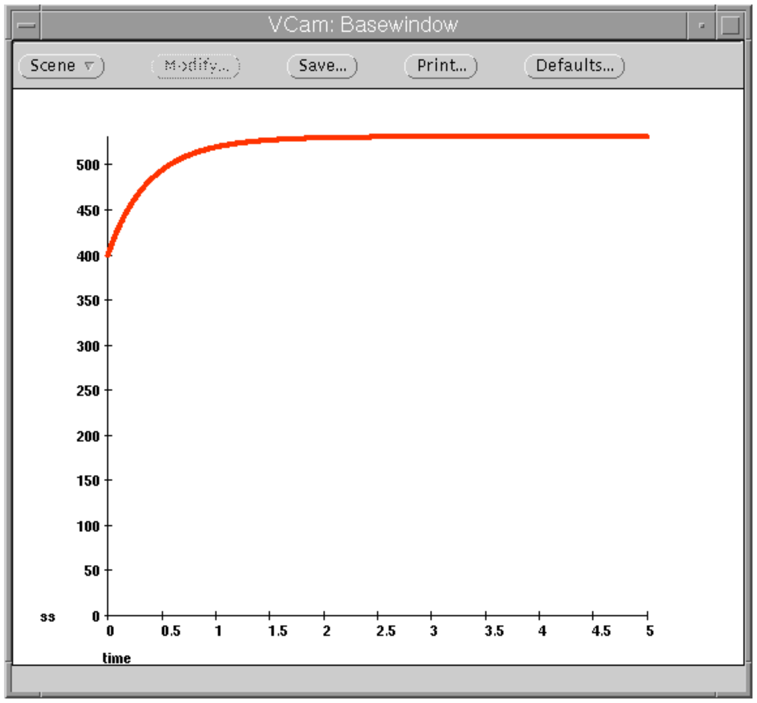


Figure 18.2: S_s, free response

18.3 Linearization, Eigenvalues, Structural Properties

Choosing the option 'Linearization' in the menu of page 127 we can get the linearized model and study it. The equilibrium point of the system is:

$$X_e = [S_o = 0.165707,\ S_s = 530.978,\ X_{BH} = 239.882]$$

PSYCO gives first the symbolic matrices of the linearized system and then the values obtained evaluating these at the equilibrium point:

$$A = \begin{bmatrix} -1073.630304 & -0.02116201945 & -1.860847195 \\ -3095.849407 & -2.064127331 & -3.910446047 \\ 2074.219103 & 0.04296531223 & -0.000001147972827 \end{bmatrix} \quad B = \begin{bmatrix} 8.934293 \\ 0 \\ 0 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \qquad D = [0]$$

Hence the linearized system is:

$$\begin{cases} \frac{dX}{dt} = \begin{bmatrix} -1073.630304 & -0.02116201945 & -1.860847195 \\ -3095.849407 & -2.064127331 & -3.910446047 \\ 2074.219103 & 0.04296531223 & -0.000001147972827 \end{bmatrix} X + \begin{bmatrix} 8.934293 \\ 0 \\ 0 \end{bmatrix} u \\ \\ y = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} X \end{cases}$$

with:

- $X \leftrightarrow (X - X_e) = \Delta X = \begin{bmatrix} \Delta S_o \\ \Delta S_s \\ \Delta X_{BH} \end{bmatrix}$
- u (the input) is the difference between Kla and its nominal value ($u = \Delta Kla = (Kla - 50)$)

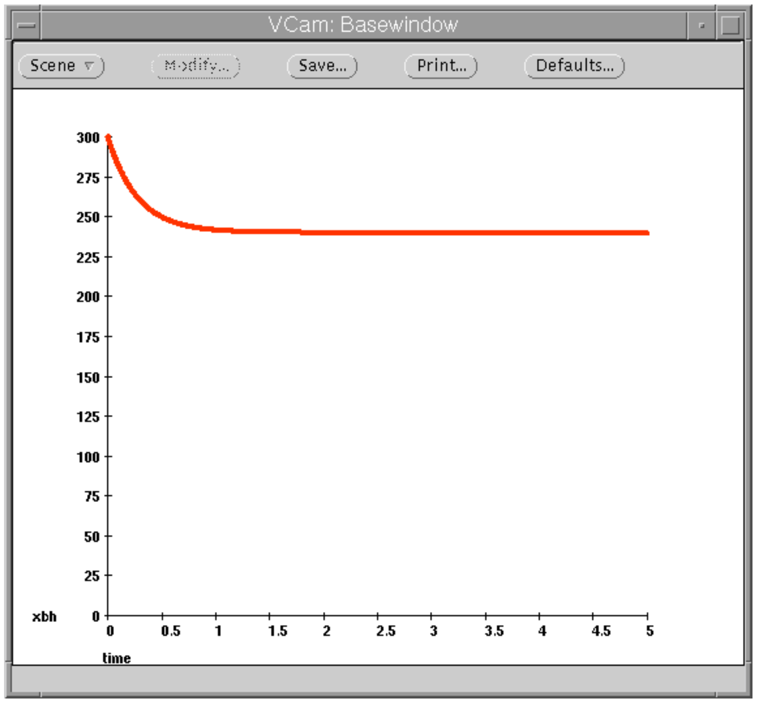


Figure 18.3: X_{BH}, free response

- y (the output) is ΔS_o

After these results PSYCO outputs the menu:

1. Eigenvalues
2. Reachability/Observability
3. Transfer function analysis/Control design
4. Linearized system time response
5. Validation
6. Exit

The eigenvalues of the linearized model are:

$$[-2.001156755, -3.609006164, -1070.084269]$$

Hence the linearized system is stable and the equilibrium point of the nonlinear system is locally stable. Choosing the 2nd option we analyze the structural properties of the system; the PSYCO output is:

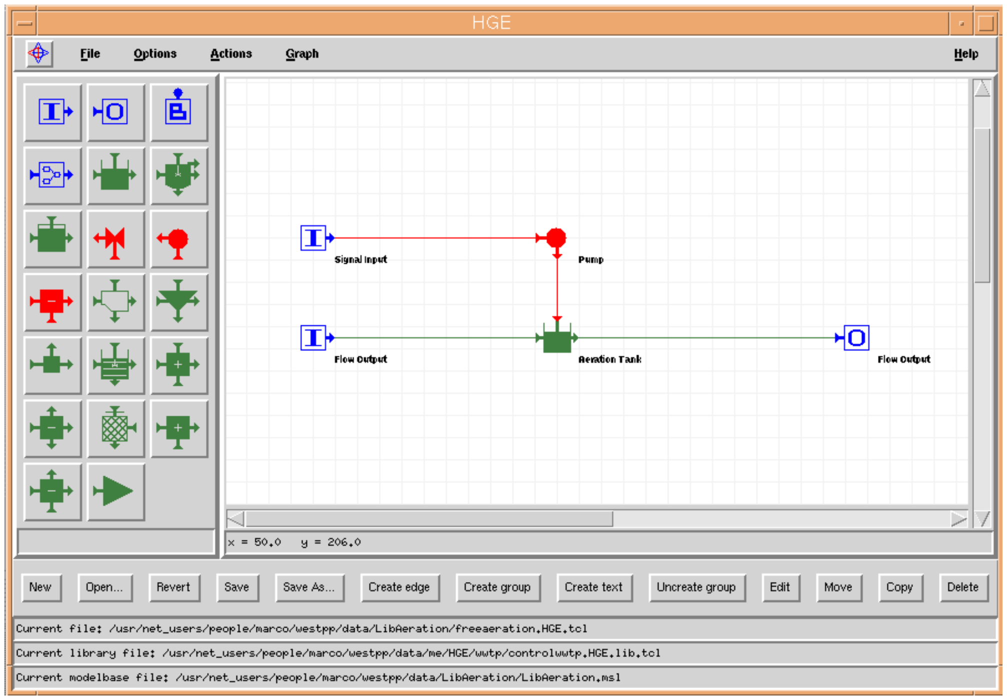


Figure 18.4: scheme for the simulation of the free behavior of the plant

$$R = \begin{bmatrix} 8.934293 & -9592.127712 & 10264499.69 \\ 0 & -27659.22569 & 29680407.92 \\ 0 & 18531.68121 & -19897362.95 \end{bmatrix}$$

$$rank(R) = 3$$

$$Completely\ reachable\ system$$

$$O = \begin{bmatrix} 1 & 0 & 0 \\ -1073.630304 & -0.02116201945 & -1.860847195 \\ 1148887.74 & 22.6839146, & 1997.944695 \end{bmatrix}$$

$$rank(O) = 3$$

$$Completely\ observable\ system$$

R is the reachability matrix and O is the observability matrix: the system is completely reachable and observable.

18.4 Transfer Function Analysis

With the 3rd option we get the transfer function of the linearized system:

$$h(s) = \frac{0.002885893925s^2 + 0.005956855841s + 0.0004848760795}{0.0003230131277s^3 + 0.3474634231s^2 + 1.941492782s + 2.496358109}$$

and then a new menu:

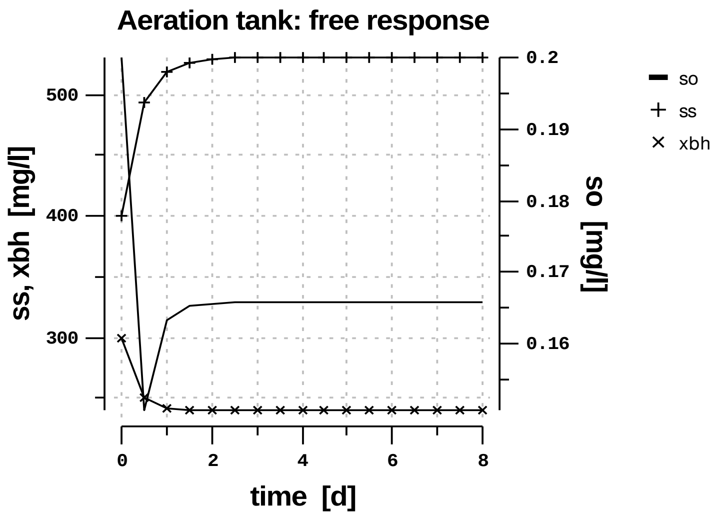


Figure 18.5: free response

1. Pole-zero representation/Gain
2. Bode, Nyquist, Nichols plots
3. Control design
4. Exit

Point 3 will be analyzed in chapter 19.
Option 1 gives:

$$Gain = 0.0001942333824$$

$$h(s) = \frac{8.934293(s+0.08488912921)(s+1.97923935)}{(s+2.001156755)(s+3.609006164)(s+1070.084269)}$$

Hence the linear system has two zeros $(-0.08488912921, -1.97923935)$ and three poles $(-2.001156755, -3.609006164, -1070.084269,$ equal to the eigenvalues) all negative and real.

The choice number 2 gives a sub-menu where it is possible to choose which diagram to plot:

- figure 18.6, 18.7: Bode diagram
- figure 18.8: Nyquist diagram
- figure 18.9: Nichols diagram

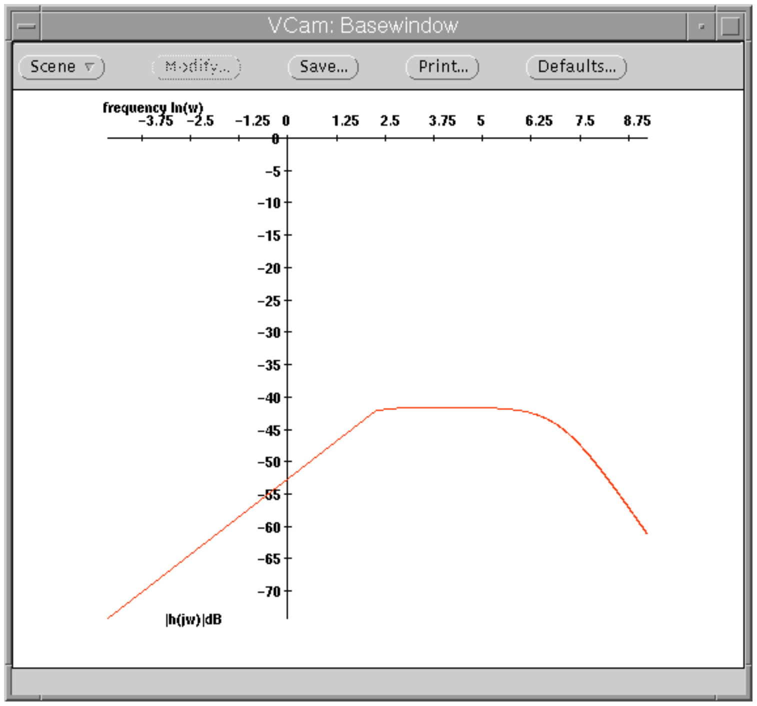


Figure 18.6: Bode amplitude plot

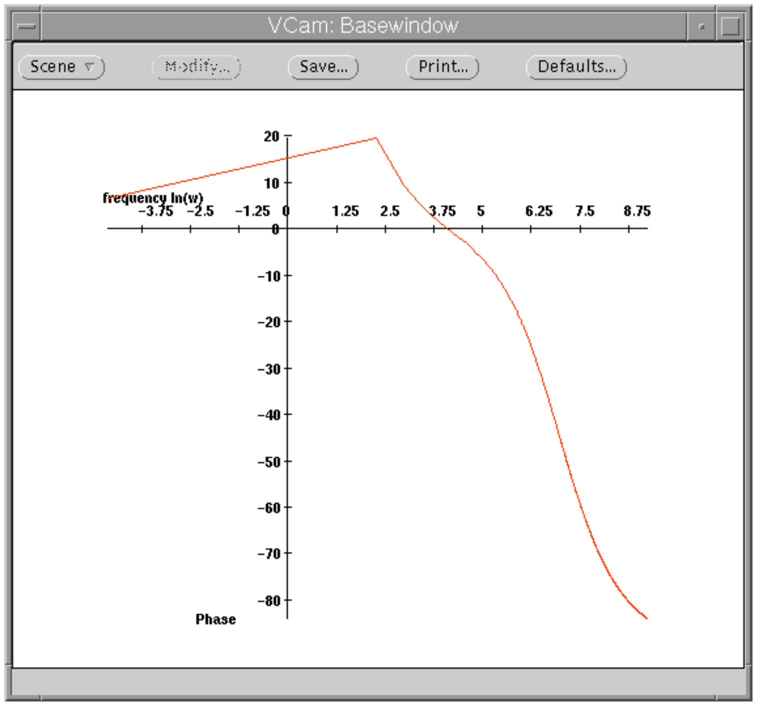


Figure 18.7: Bode phase plot

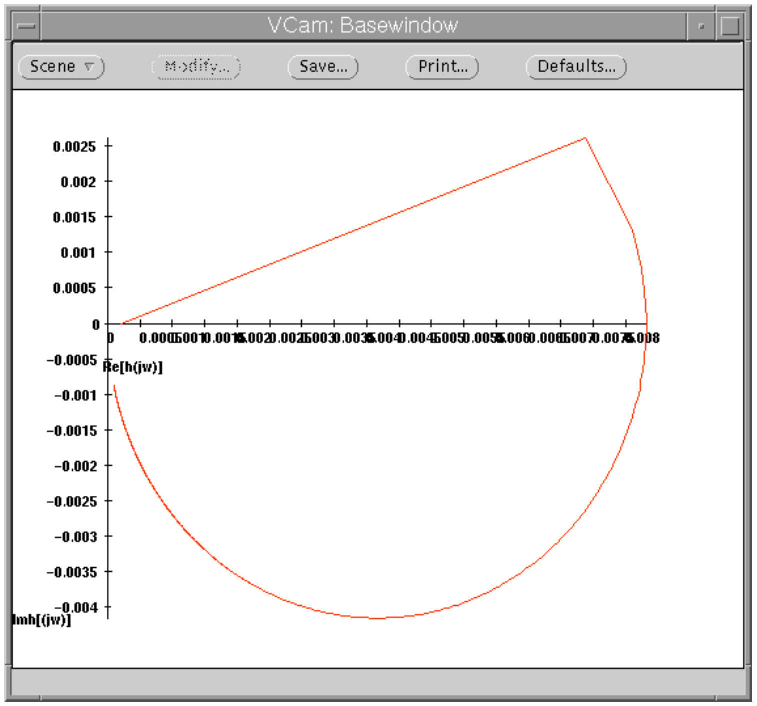


Figure 18.8: Nyquist plot

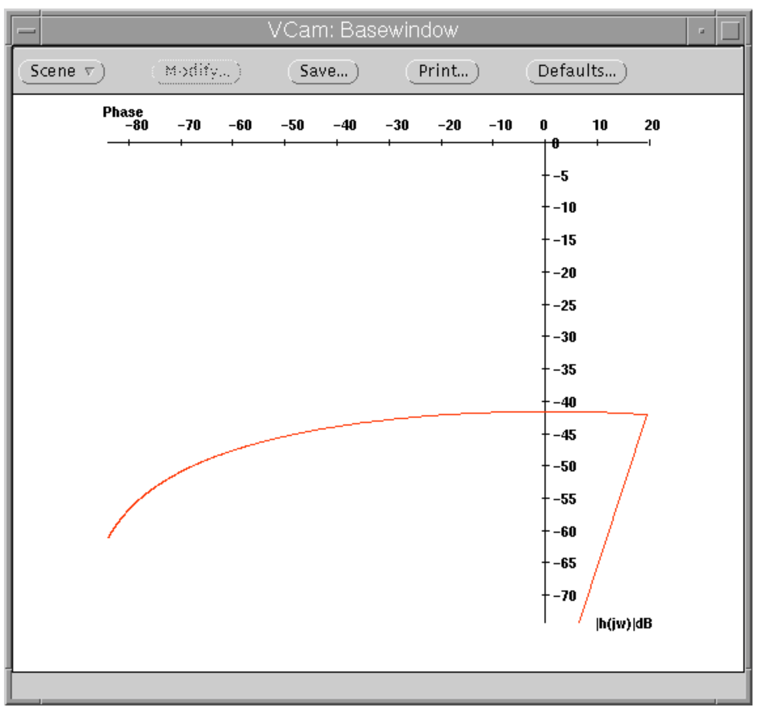


Figure 18.9: Nichols plot

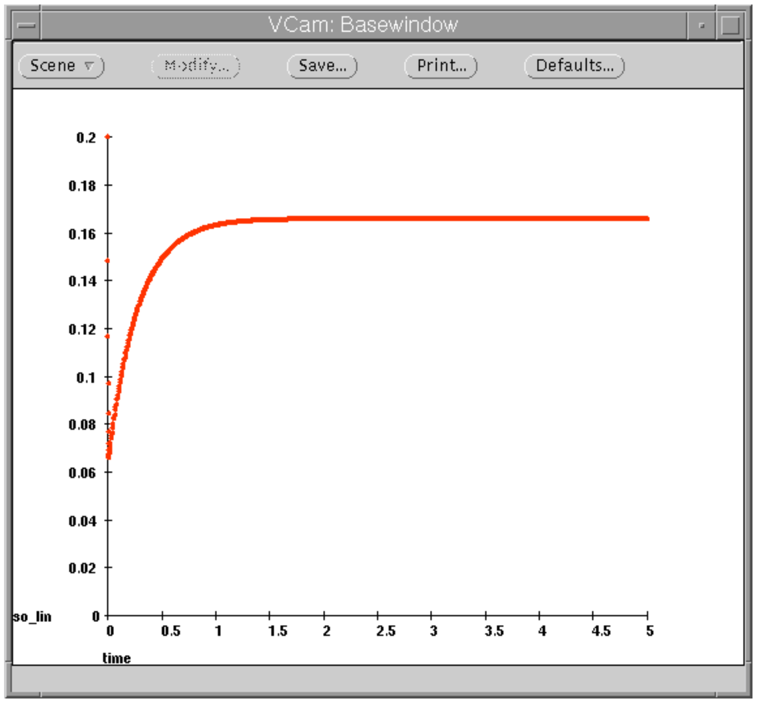


Figure 18.10: $S_o(t)$, free response plot

18.5 Linearized System Time Response – Validation

Let us choose the 4th option in the menu of page 130. After choosing the initial conditions, the simulation time, the kind of input to sum over the nominal value we obtain the plot of the linearized system. Furthermore we can choose which variable (state or output) to plot and whether we want a plot related to the equilibrium point or to the zero. Figures 18.10, 18.11 and 18.12 are an example for a free response with Kla and initial conditions as in section 18.2.

The 5th point in the menu of page 130 gives the possibility of comparing the behavior between the nonlinear system and the linearized one. Choosing the same options as above we get the mean square error between the variables chosen to be compared. Furthermore a qualitative validation is obtained by plotting both the variables together in the same diagram. Figures 18.13 , 18.14, 18.15 show this plot for Kla and initial conditions like in section 18.2. The mean square error is 0.00441744537 for S_o, 0.132983732 for S_s and 0.07656611128 for X_{BH}.

From these data we observe that the linear system is a good approximation to the nonlinear one and hence the linear system can be used to represent, near the equilibrium point, the nonlinear system.

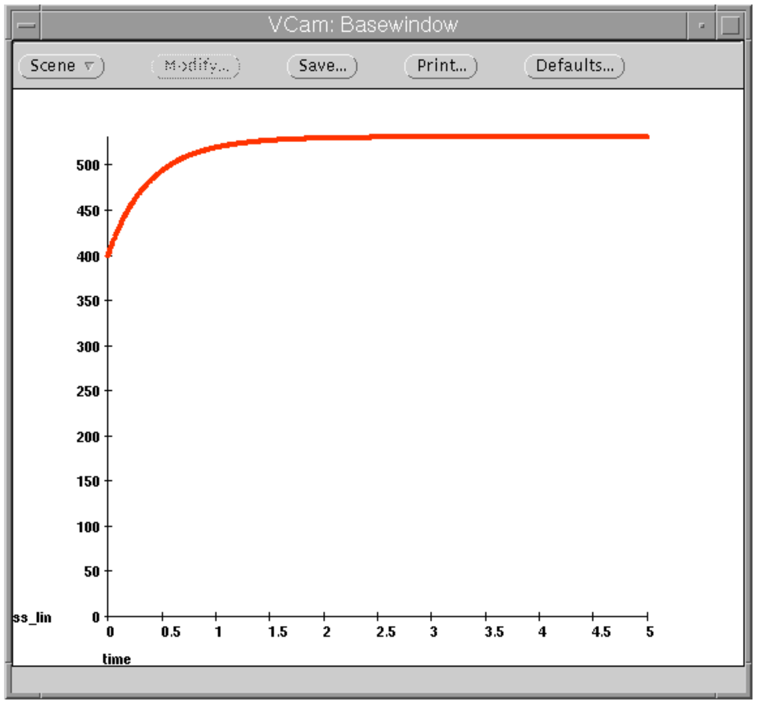


Figure 18.11: $S_s(t)$, free response plot

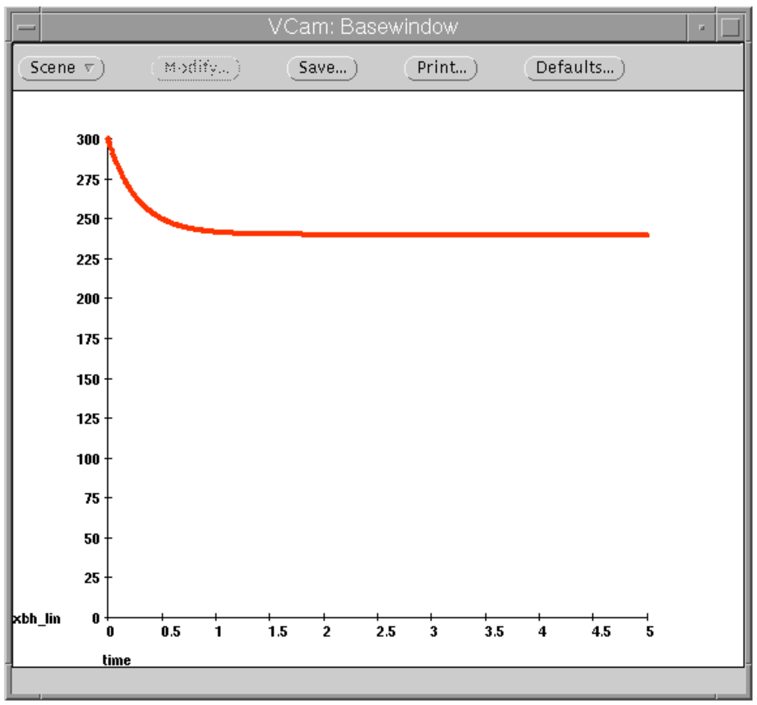


Figure 18.12: $X_{BH}(t)$, free response plot

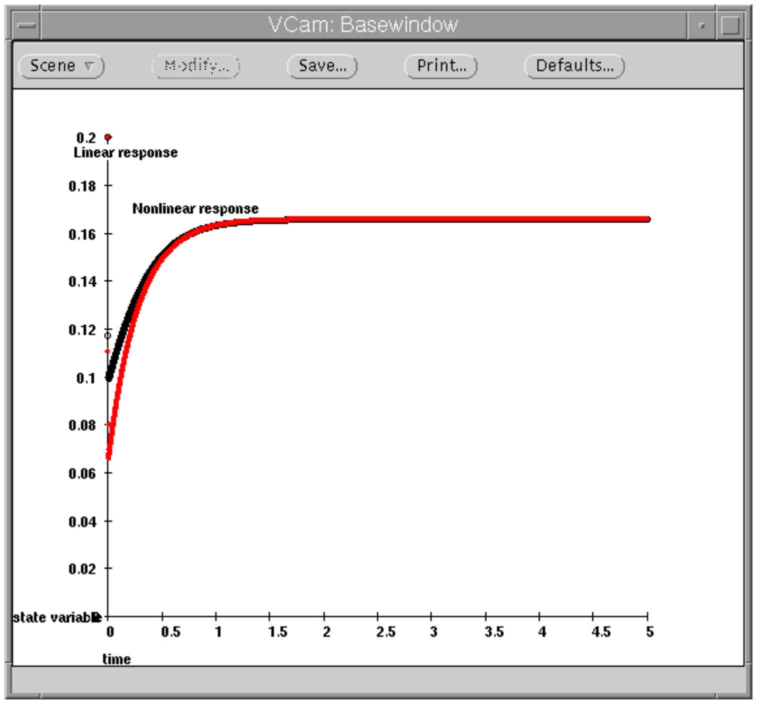


Figure 18.13: $S_o(t)$, validation

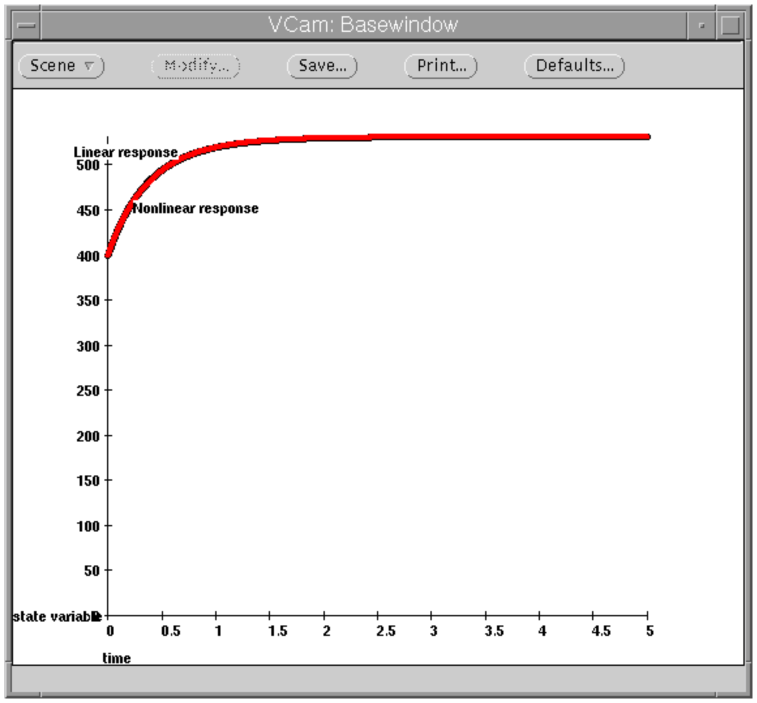


Figure 18.14: $S_s(t)$, validation

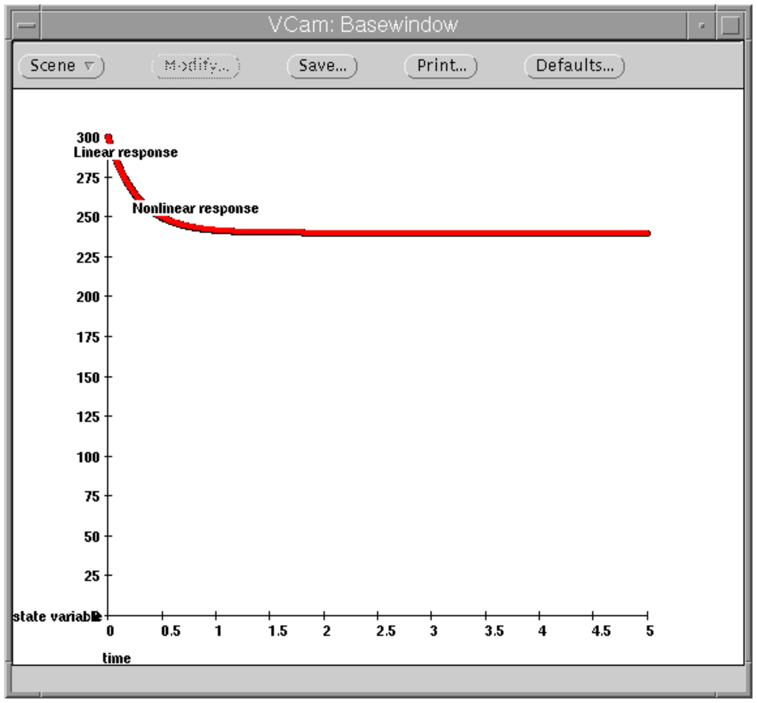


Figure 18.15: $X_{BH}(t)$, validation

19
Aeration Tank: Control Design

19.1 Stability Analysis Tools

Choosing the option “Control design” in the menu of page 131, PSYCO gives the possibility to build a controller.
The menu gives the choice between:

- Stability analysis tools
- Controllers
- Exit

If we want to study the feedback stability properties we can choose the 1st option. The root locus equation is:

$$0.0004848760795k + 1.941492782s + 0.005956855841ks + 0.3474634231s^2$$

$$+0.0003230131277s^3 + 0.002885893925ks^2 + 2.496358109$$

Figure 19.1 shows the root locus plot.

PSYCO also gives the opportunity to get the gain and phase margins. In our example it will not return anything because of the particular behavior of the system (see the Nyquist plot, figure 18.8).

19.2 Controllers

PSYCO can build controllers such as:

- PID
- Direct Design
- Robust Control
- User Controller

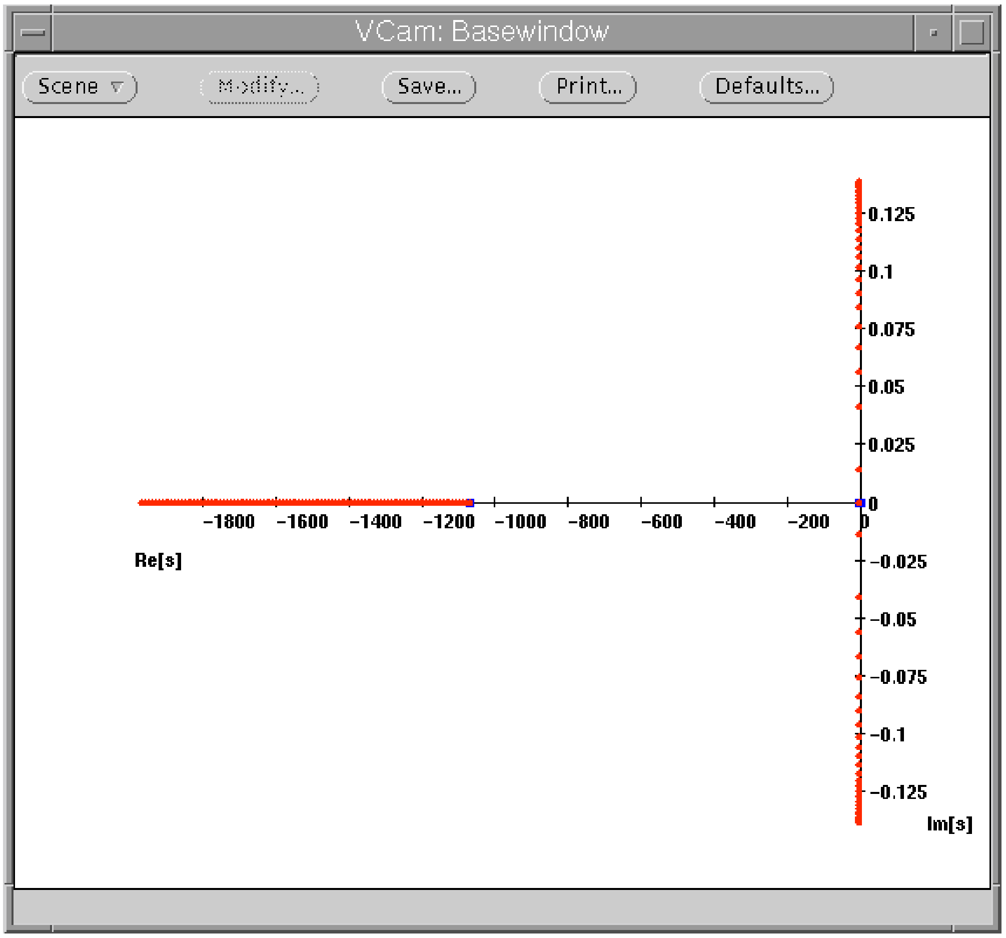


Figure 19.1: Root Locus

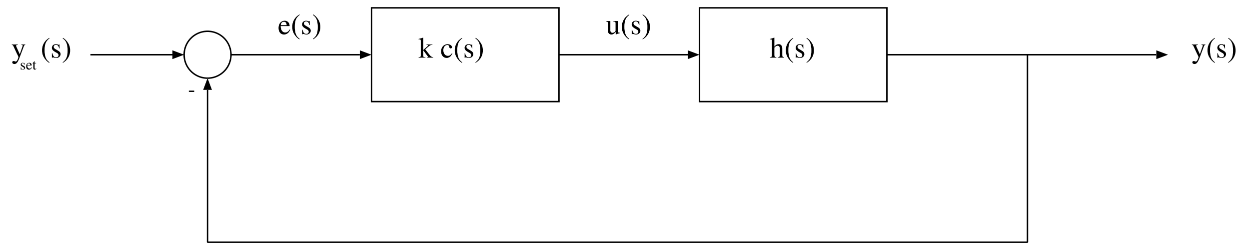


Figure 19.2: Feedback control scheme

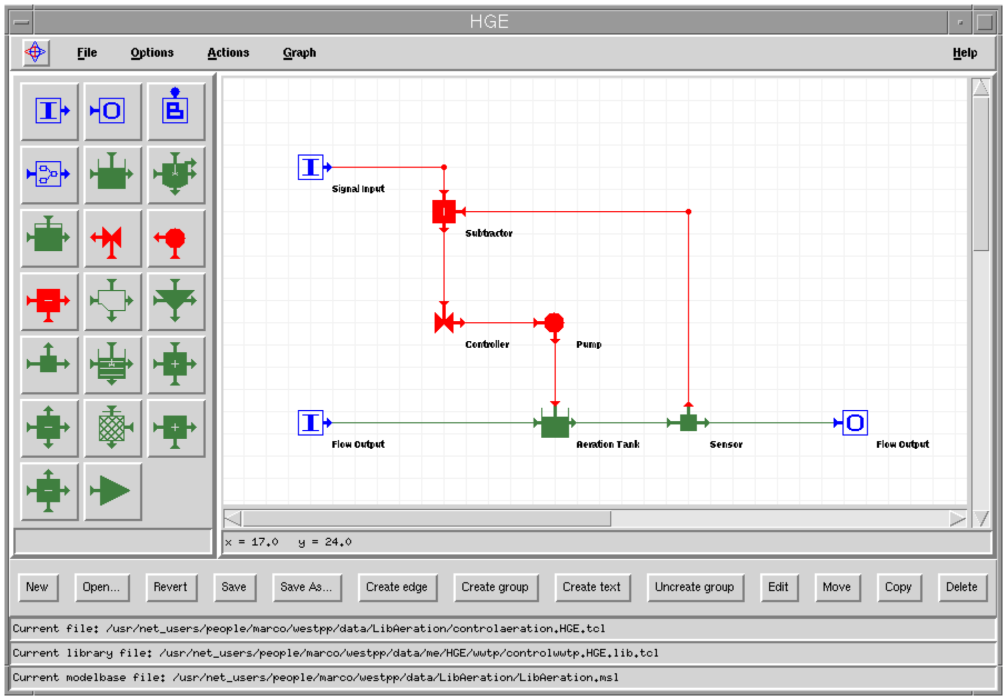


Figure 19.3: HGE controlled scheme for 'classic' feedback system

Apart from robust control, all these controllers are based on the 'classic' feedback scheme (see figure 19.2). PSYCO will find the controllers and then will analyze the controller, the controlled forward chain, and the closed loop.
For the simulation of the feedback system, WEST++ can be used. For this purpose it is necessary to have a state representation of the controllers. The latter can be obtained with PSYCO: choosing option 2 in the menu in page 127, given a transfer function C(s), it returns the canonical reachable realization of C(s). This can then be inserted in a MSL-USER model and run in WEST++. Figure 19.3 shows the scheme in the HGE window (modeling environment) used for this purpose.

Robust control will be dealt within the section 19.3. User control is an option that gives to the user the opportunity to analyze a controller found by others tools.

19.2.1 PID controllers

We choose the design frequency (ω_0) equal to $1day^{-1}$. PSYCO gives the open loop amplitude and phase at $\omega_0 = 1$, and plots the Nyquist diagram (useful to design this controller). We set the forward chain amplitude equal to 0.2 and the phase equal to $165°$. The parameters found are:

- $K_p = 9.042821354$
- $T_i = 0.08958675716$

- $T_d = 0.04479337858$

With WEST++ we have made two simulations:

1. figures 19.4, 19.5: the flow rate is constant (i.e. $Q/V = 2day^{-1}$)
2. figures 19.6, 19.7: the flow rate is: $Q/V = 2 + 0.2\sin(4t) + 0.1 * random$ ($random$ is a function that returns random values in [-1;1])

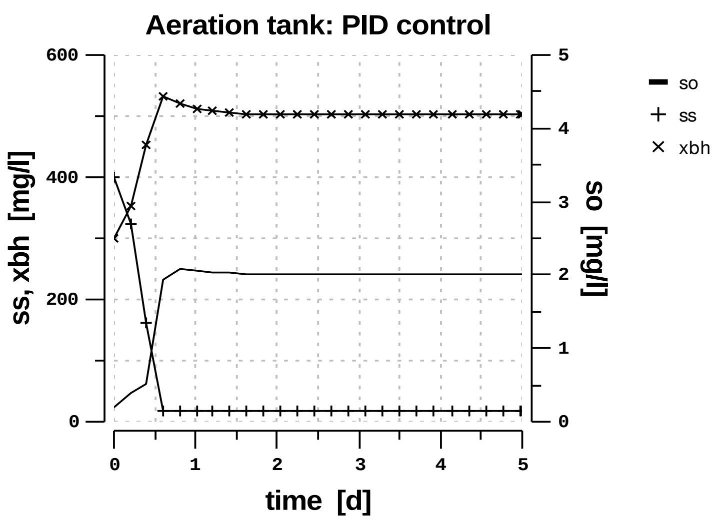


Figure 19.4: State plot, simulation $n°1$

19.2.2 Direct Design Controller

We set the closed loop transfer function equal to

$$\frac{1}{s+1}$$

PSYCO gives the controller:

$$C(s) = \frac{0.0003230131277s^3 + 0.3474634231s^2 + 1.941492782s + 2.496358109}{0.002885893925s^3 + 0.005956855841s^2 + 0.0004848760795s}$$

The forward chain becomes:

$$L(s) = C(s)h(s) = \frac{1}{s}$$

The simulations are with the same conditions as above. Figures 19.8, 19.9, 19.10, 19.11 show the behavior.

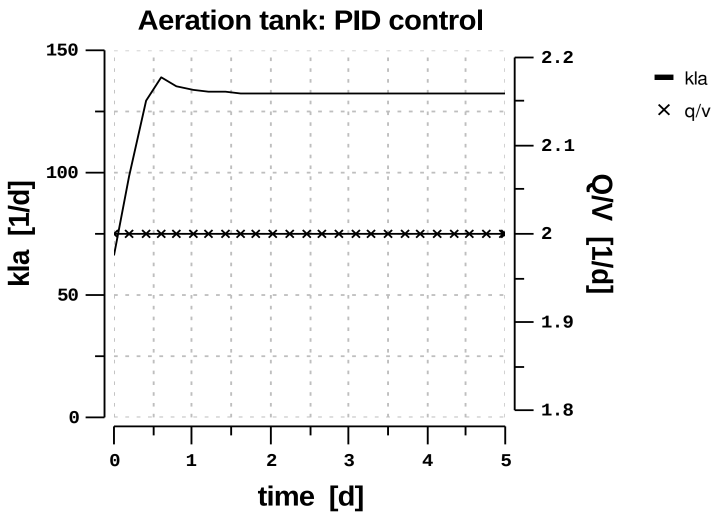


Figure 19.5: kla, Q/V plot, simulation $n°1$

19.3 Robust Control

This robust control is based on the IMC scheme (see figure 19.12).
The IMC controller is:

$$C(s) = \frac{0.1119282746(s + 2.001156754)(s + 3.609006166)(s + 1070.084269)}{(1 + as)(s + 0.0848891292)(s + 1.97923935)}$$

The parameter a has to be fixed on line, by the simulation. The Model in figure 19.12 is:

$$h(s) = \frac{8.934293(s + 0.08488912921)(s + 1.97923935)}{(s + 2.001156755)(s + 3.609006164)(s + 1070.084269)}$$

To simulate this, the HGE scheme is more complex (figure 19.13).
Figures 19.14, 19.15, 19.16, 19.17 show the behavior of the robust controlled system with the same conditions as in page 142, and with the value of the parameter a equal to 0.1.
Figures 19.18 and 19.19 show the 2° simulation of the system with $a = 1$.
The value of $a = 0.1$ turns out to be a good compromise between performance and robustness [7].

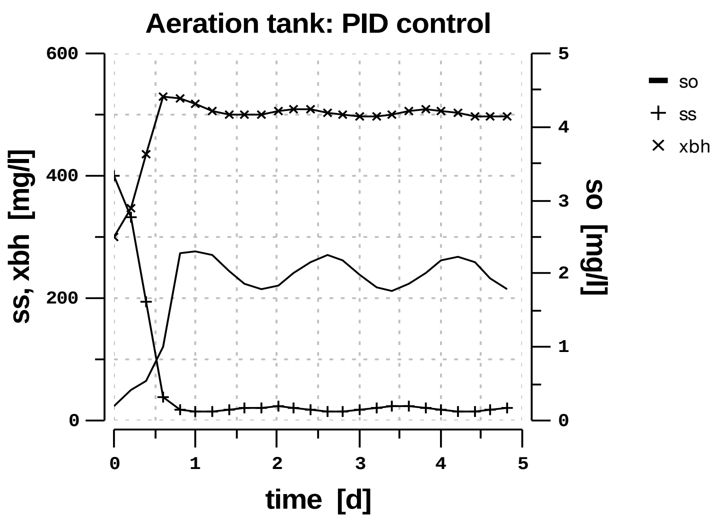


Figure 19.6: State plot, simulation $n°2$

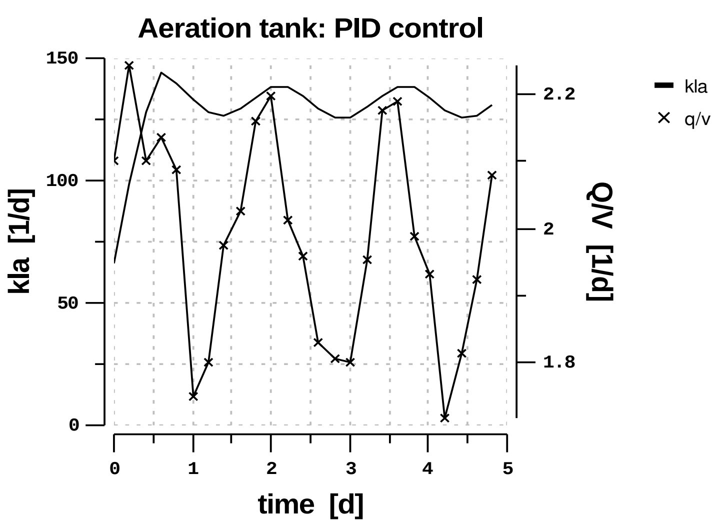


Figure 19.7: kla, Q/V plot, simulation $n°2$

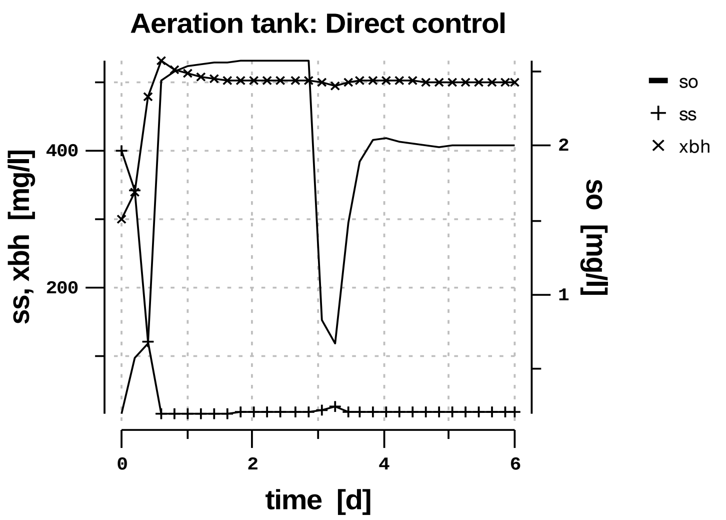


Figure 19.8: State plot, simulation $n°1$

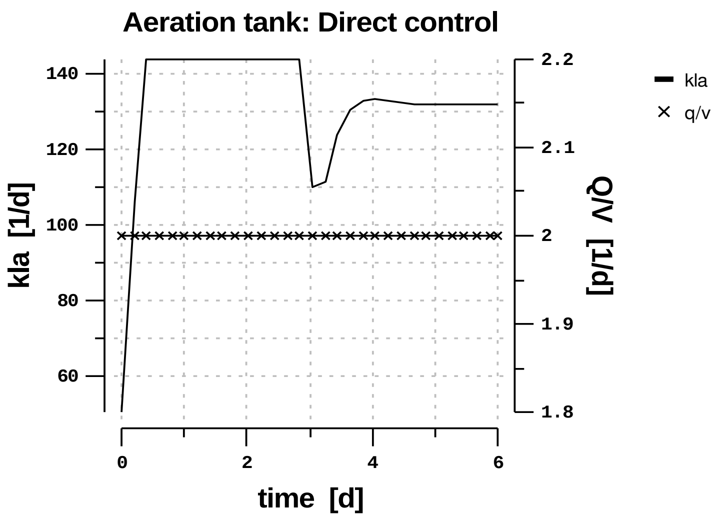


Figure 19.9: kla, Q/V plot, simulation $n°1$

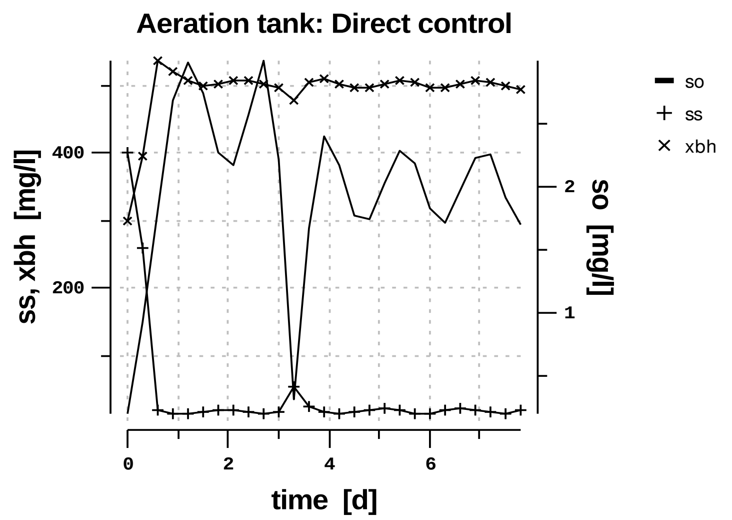


Figure 19.10: State plot, simulation $n^{\circ}2$

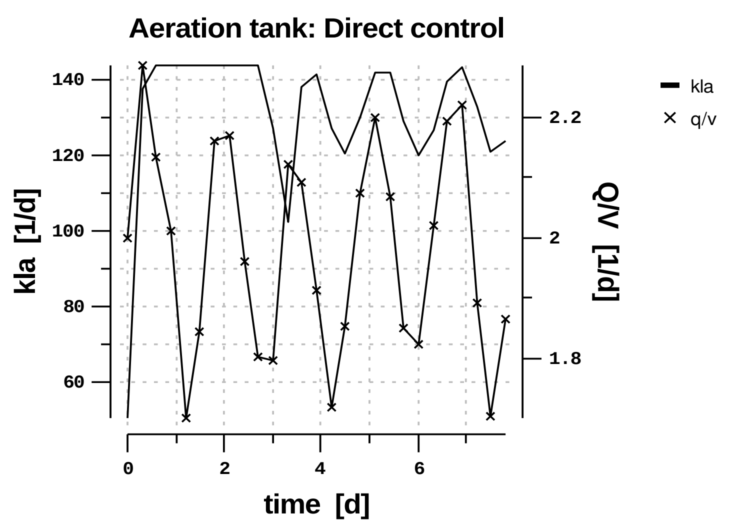


Figure 19.11: kla, Q/V plot, simulation $n^{\circ}2$

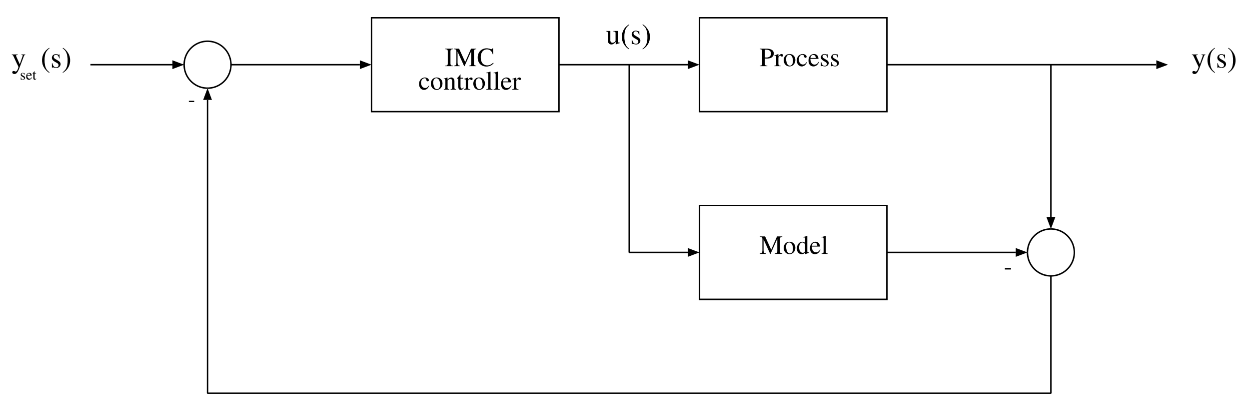


Figure 19.12: IMC control scheme

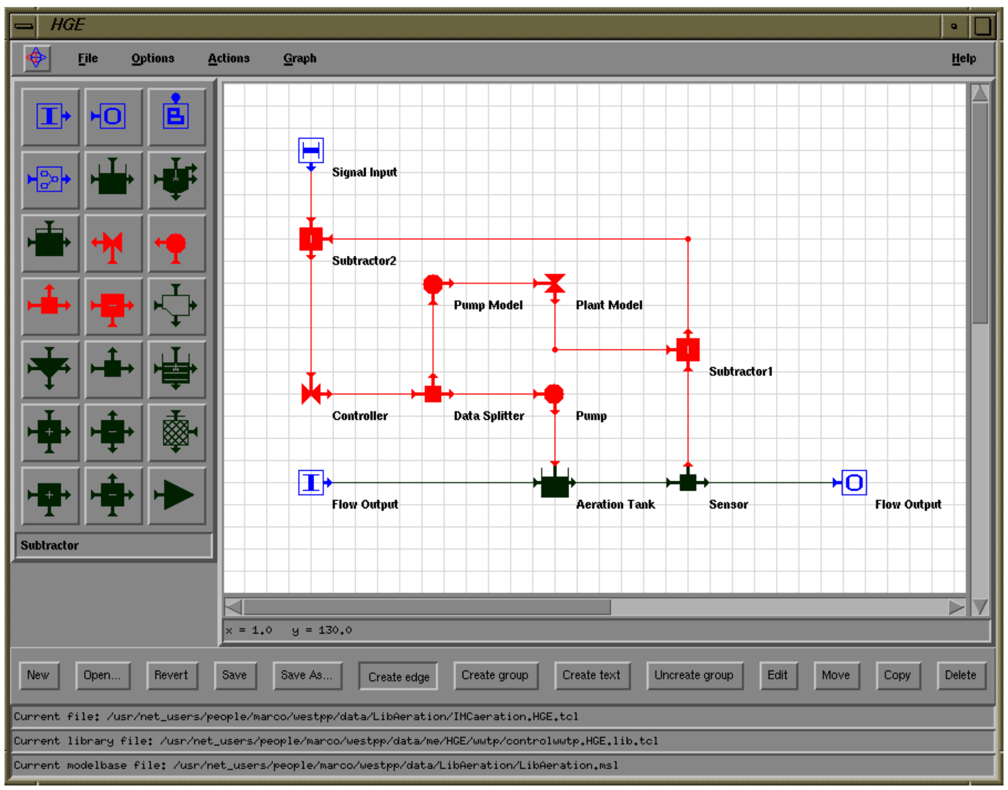


Figure 19.13: HGE controlled scheme for IMC system

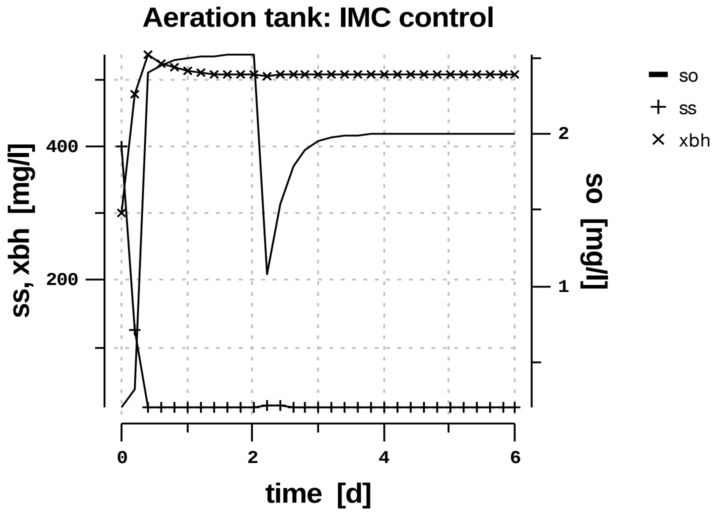


Figure 19.14: State plot,$a = 0.1$, simulation $n°1$

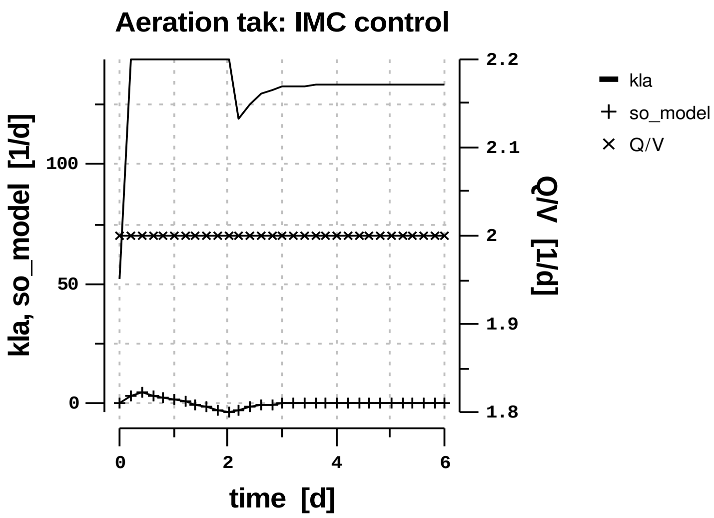


Figure 19.15: kla, Q/V, model output plot,$a = 0.1$, simulation $n°1$

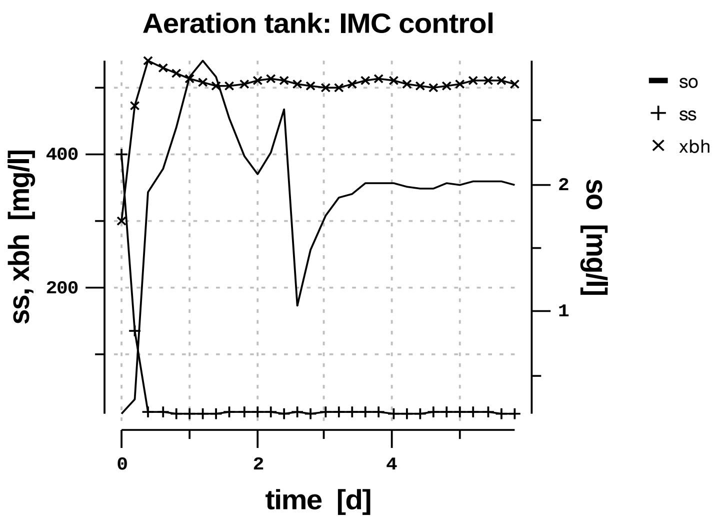


Figure 19.16: State plot,$a = 0.1$, simulation $n°2$

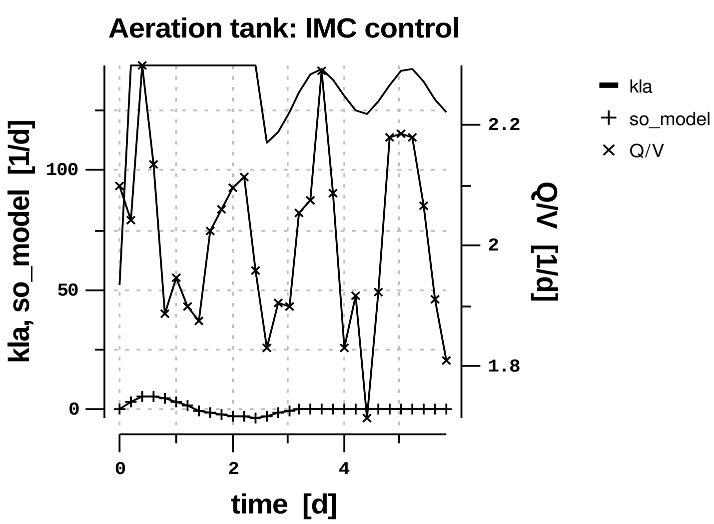


Figure 19.17: kla, Q/V, model output plot,$a = 0.1$, simulation $n°2$

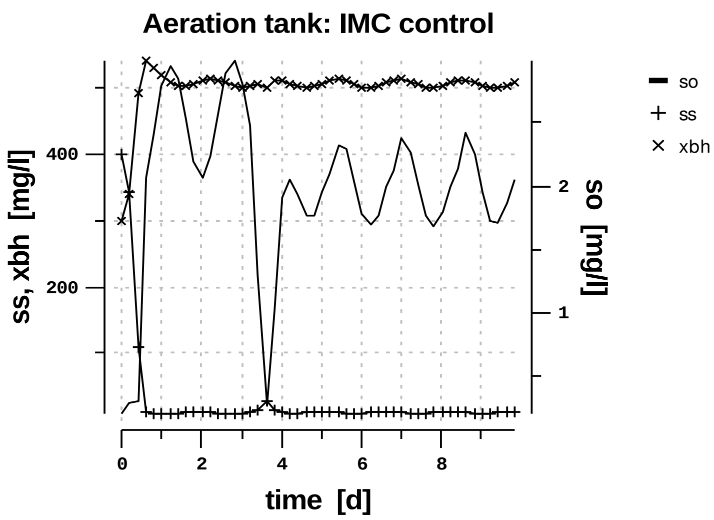


Figure 19.18: State plot, $a = 1$, simulation $n°2$

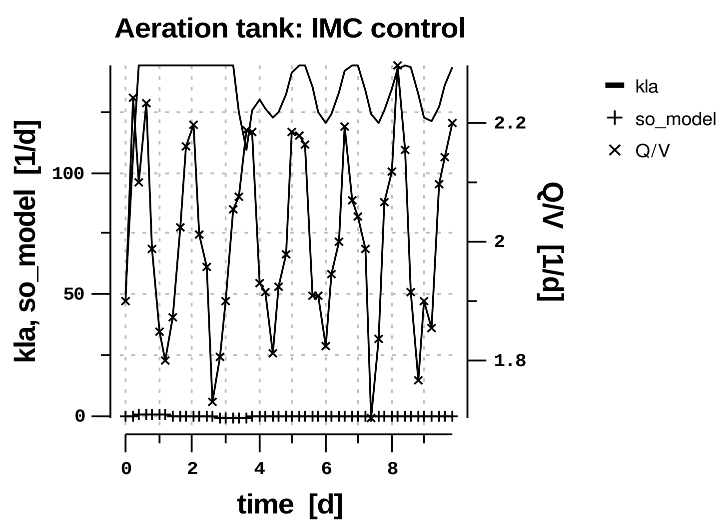


Figure 19.19: kla, Q/V, model output plot, $a = 1$, simulation $n°2$

Conclusion

PSYCO is the result of research, study and computer programming which kept us intellectually busy for a long period. Several hurdles has been met, but we always tackled them as a challenge: it was not rare to work until late even during the weekend, as long as a solution of a problem was not reached. Actually, fixing a problem was nothing but an excuse to have a nice glass of Belgian beer.
We are rather proud of our work and we hope that PSYCO can be useful for users interested in analysis and design of plants from a control engineering point of view.
Of course, PSYCO is a completely working software, but we can already look ahead: extensions and improvements can be foreseen. First of all, some meaningful control technique and tool could be added. The current version of PSYCO considers continuous time systems, but a similar system could be developed for discrete time systems and then connections of hybrid systems could be considered as well.
Currently, PSYCO is written in MuPAD and so its performance is not optimal; that means, since using symbolic manipulation is very CPU consuming and MuPAD is a general purpose tool, a translation to C++ code would allow better performance and last, but not least, a complete integration with the WEST++ environment.
It is claimed in this report that symbolic analysis offers many advantages over numerical approaches (as implemented in Matlab and Simulink [9]). A quantification of this statement should be made in the future.
PSYCO has been developed mainly at the BIOMATH department of the Gent University (Belgium). This was possible thanks to a Socrates/Erasmus student exchange programme between Florence University and Gent University; that allowed us to spend a period of our life in a foreign country. It has been a very positive experience and not only from a professional point of view. Living abroad taught us different habits, that can be appreciated only in their context.
During this period, many people supported us both professionally and morally.
First of all, we would like to thank our supervisor at the BIOMATH Dept. Hans Vangheluwe: he drove us along all our work, listening to our problems and giving us precious advice; we appreciate him especially for his energy and his friendliness. A special thanks to prof. Stefano Marsili Libelli who was the enthusiastic promoter of this project and gave us the opportunity to go to the University of Gent; even at a distance, he kept always in touch with us giving his worthwhile opinions based on his deep experience.
Furthermore, we would like to thank the BIOMATH department and particularly the simulation lab, our families, all the people known at “Home Merlijin” and our friends.

Gian Lorenzo Lucchetti
Marco Zoccadelli

A
MSL-USER code

The following code is a MSL-USER library built to simulate the aeration tank behavior.

```
// AtomicModelbase.msl (syntax: MSL-USER-3.1) HV     10/3/1998
//                                           Khier  25/3/1998
//                                           HV     28/3/1998
//                                           Henk   09/4/1998
//                                           Khier  22/4/1998
//
// lib.generic.msl (syntax: MSL-USER-3.1) HV  20/ 2/1998
//
// Contains generic declarations for the modelling of
// dynamic (DAE based) physical systems
//

// Builtin types are the only types for which
// an empty signature is allowed.
// During bootstrapping, the builtin type names
// are loaded into the outermost type namespace.
// The semantics of these types is given implicitly.

// Builtin atomic types


 TYPE Generic "builtin: type variable";
 // The Generic type is a "type variable". It will unify with any
 // other type; any type is a sub-type of Generic which implies any
 // object can be an instance of type Generic.

 TYPE Integer "builtin: positive and negative Natural Numbers";

 TYPE Real "builtin: Real numbers";

 TYPE Char "builtin: ASCII character";

 TYPE String
  "builtin: Char* (implemented as atomic type for efficiency reasons)";

 TYPE Bottom "builtin: bottom type" = ENUM {null};
 // The Bottom type is a sub-type of any other type.
```

```
 // By virtue of this, "null", the only object of
 // type Bottom, can be used to denote an unassigned value
 // for objects of any type.

 TYPE Boolean "builtin: Logic type" = ENUM {True, False};

// Builtin composite types


 TYPE TypeDeclarationType
 "builtin: type of TYPE declaration statement";

 TYPE ClassDeclarationType
 "builtin: type of CLASS declaration statement";

 TYPE ObjectDeclarationType
 "builtin: type of OBJ declaration statement";

 TYPE DeclarationType
 "type of a declaration (TYPE, CLASS, or OBJ) statement"
 = UNION {TypeDeclarationType, ClassDeclarationType, ObjectDeclarationType};

 TYPE ExpressionType
 "builtin: type of expressions";

 TYPE EquationType
 "builtin: type of equations";

 TYPE GenericIntervalType
 "
  Generic Interval. Only meaningful if used
  to specialise with endpoints of a type for which
  an order relation is defined.
 "
 = RECORD
   {
    lowerBound: Generic;
    upperBound: Generic;
    lowerIncluded: Boolean;
    upperIncluded: Boolean;
   };

 TYPE RealIntervalType "Interval of real numbers"
 SUBSUMES GenericIntervalType =
 RECORD
 {
  lowerBound: Real; // Real is sub-type of Generic
  upperBound: Real; // Real is sub-type of Generic
  lowerIncluded: Boolean;
  upperIncluded: Boolean;
 };


// type declarations for physical systems
//

 TYPE UnitType
 "The type of physical units. For the time being, a string"
= String;

 TYPE QuantityType
 "The different physical quantities. For the time being, a
```

```
string" = String;

 TYPE CausalityType
 " Causality of entities:
    CIN: input (cause) only
    COUT: output (consequence) only
    CINOUT: input and output (cause and consequence) are allowed
 "
 = ENUM {CIN, COUT, CINOUT};

 TYPE PhysicalNatureType
 "The nature of physical variables
  FIELD is used (in the physicalDAE context) to denote
  parameters and constants
 "
 = ENUM {ACROSS, THROUGH, FIELD};

 TYPE PhysicalQuantityType
 "The type of any physical quantity"
 =
 RECORD
 {
  quantity   : QuantityType;
  unit       : UnitType;
  interval   : RealIntervalType;
  value      : Real;
  causality  : CausalityType;
  nature     : PhysicalNatureType;
 };

 // Formalism independent model stuff


 TYPE InterfaceDeclarationType
 "declarations within an interface" = DeclarationType;

 TYPE ParameterDeclarationType
 "declarations within parameter section" = DeclarationType;

 TYPE ModelDeclarationType
 "declarations within sub_models section" = DeclarationType;

 TYPE CouplingStatementType
 "parameter coupling and connect() statements" = EquationType;

 TYPE GenericModelType
 "The signature of the generic part of any (whatever the formalism) model"
 =
 RECORD
 {
  comments   : String;
  interface  : SET_OF (InterfaceDeclarationType);
   // declared objects must be interfaces
  parameters : SET_OF (ParameterDeclarationType);
   // declared objects must be parameters
 };

 TYPE CoupledModelType "The signature of a coupled (network) model"
 EXTENDS GenericModelType WITH
 RECORD
 {
  sub_models : SET_OF (ModelDeclarationType);
```

```
  coupling   : SET_OF (CouplingStatementType);
 };

 TYPE DAEModelType
 "The signature of a Differential Algebraic Equation (DAE) model
  within DAEModelType models, connect() has the following
  (flattening) semantics:
    quantity and unit are checked for equality
    equations are generated to equal (=) all algebraic and state variables
    all other labels are ignored
 "
 EXTENDS GenericModelType WITH
 RECORD
 {
  independent : SET_OF (ObjectDeclarationType); // independent variable (time)
  state       : SET_OF (PhysicalQuantityType); // variables
                                    // those variables occurring in
                                    // DERIV(v, [t]) statements are
                                    // derived state variables
  initial     : SET_OF (EquationType);
  equations   : SET_OF (EquationType);
  terminal    : SET_OF (EquationType);
 };

 TYPE PhysicalDAEModelType
 "within physicalDAEModelType models, connect() has the
following
  (flattening) semantics:
    quantity and unit are checked for equality
    quantity and unit are checked for equality
    equations are generated to equal (=) all across variables
    equations are generated to sum all through variables to zero
    all other labels are ignored
 "
 = DAEModelType;

 // The meaning of TYPE and CLASS extension
 //
 //   The extension signature and the original signature must be
 //   of the same type. If the types are equal, extension has a well-defined
 //   meaning (concatenation, fail if overlap), currently only for RECORD and
 //   SET_OF type signatures.
 //

// End of lib.generic.msl

// Some type declarations

  CLASS MassFlux = PhysicalQuantityType:= {: nature <- "ACROSS" :};
  // := {: nature <- "THROUGH" :};

  CLASS Time
  "The type of time"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity  <- "Time";
    unit      <- "day";
    interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF;:};
  :};

  CLASS Yield
  "Yield"
```

```
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity <- "Yield";
    unit     <- "";
    interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF:};
  :};

  CLASS GrowthRate
  "GrowthRate"
  SPECIALISES PhysicalQuantityType :=
  {:
    quantity <- "GrowthRate";
    unit     <- "";
    interval  <- {: lowerBound <- 0; upperBound <- PLUS_INF:};
  :};

  TYPE Components
  " The biological components considered in the WWTP model"
  = ENUM {Q_V,S_O,S_S,X_BH}; // Q_V is not a component,
                             // it is the flow rate !!!

  OBJ NrOfComponents
  " The number of components"
  : Integer := Cardinality(Components);


  CLASS WWTPTerminal
  "The variables which are passed between WWTP model building blocks"
  = MassFlux[NrOfComponents;];

  CLASS inWWTPTerminal SPECIALISES WWTPTerminal; //used to indicate inflow
  CLASS outWWTPTerminal SPECIALISES WWTPTerminal; //used to indicage outflow

  CLASS Signal SPECIALISES PhysicalQuantityType;

  OBJ comp_index "Temporary iteration variable" : Integer;
  OBJ reaction_index "Temporary iteration variable" : Integer;
  OBJ in_comp_index "Temporary iteration variable" : Integer;
  OBJ out_comp_index "Temporary iteration variable" : Integer;
  OBJ terminal "Temporary iteration variable" : WWTPTerminal;
  OBJ in_terminal "Temporary iteration variable" : WWTPTerminal;
  OBJ out_terminal "Temporary iteration variable" : WWTPTerminal;


// Model Blocks

CLASS flowInputGenerator
  (* class = "input"; category = "simple" *)
  "flowinputgenerator"
  SPECIALISES PhysicalDAEModelType :=
 {:
    interface <-
    {
    OBJ Outflow (* terminal = "out" *) "outflow: Q_V, S_O, S_S, X_BH" :
          outWWTPTerminal  := {: causality <- "COUT" :};
    };

    parameters <-
    {
    OBJ S_Oin:  PhysicalQuantityType := {: value <- 0 :};
    OBJ S_Sin:  PhysicalQuantityType := {: value <- 1000 :};
    OBJ X_BHin:  PhysicalQuantityType := {: value <- 0 :};
```

```
    // Q/V = Step+Asin(omega*t)+noise
    OBJ Step " Q/V = Step+Asin(omega*t)+noise_fraction*random"
        :  PhysicalQuantityType := {: value <- 2 :};
    OBJ A:  PhysicalQuantityType :=  {: value <- 0 :};
    OBJ omega:  PhysicalQuantityType :=  {: value <- 0 :};
    OBJ noise_fraction "noise fraction" :
        PhysicalQuantityType :=  {: value <- 0 :};
    };

    independent <-
    {
      OBJ t "independent variable time": Time
    };

    equations <-
    {
    interface.Outflow[Q_V] =
      parameters.Step + parameters.A*sin(parameters.omega*independent.t)
      + parameters.noise_fraction*my_random(independent.t);

    interface.Outflow[S_O] = parameters.S_Oin ;
    interface.Outflow[S_S] = parameters.S_Sin ;
    interface.Outflow[X_BH] = parameters.X_BHin ;
    };

 :};

CLASS flowOutput
  (* class = "output"; category = "simple" *)
  "Output flow"
  SPECIALISES
  PhysicalDAEModelType :=
  {:
    interface <-
    {
      OBJ Inflow (* terminal = "in" *) "inflow" :
        inWWTPTerminal := {: causality <- "CIN" :};
    };
  :};

CLASS signalInputGenerator
  (* class = "input"; category = "simple" *)
  "signalinputgenerator"
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
    {
    OBJ Outsignal  (* terminal = "out" *) "outsignal: Step + Asin(omega*t)" :
          Signal  := {: causality <- "COUT" :};
    };

    parameters <-
    {
    OBJ Step:  PhysicalQuantityType := {: value <- 2 :};
    OBJ A:  PhysicalQuantityType :=  {: value <- 0 :};
    OBJ omega:  PhysicalQuantityType :=  {: value <- 0 :};
    };

    independent <-
    {
      OBJ t "independent variable time": Time
```

```
    };

    equations <-
    {
    interface.Outsignal =
      parameters.Step + parameters.A*sin(parameters.omega*independent.t);
    };
 :};

CLASS Pump
   (* class = "controller"; category = "simple" *)
  "pump"
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
    {
    OBJ Insignal (* terminal = "in" *) "insignal" :
          Signal := {: causality <- "CIN" :};
    OBJ Outsignal (* terminal = "out" *) "outsignal" :
          Signal := {: causality <- "COUT" :};
    };

    parameters <-
    {
     OBJ u0 "Initial value for manipulated variable (no error action)" :
        PhysicalQuantityType := {: value <- 50 :};
     OBJ max "Max value of the pump output" :
        PhysicalQuantityType := {: value <- 144 :};
    };

    equations <-
    {
     interface.Outsignal =
           IF (interface.Insignal > -parameters.u0)
              THEN IF (interface.Insignal + parameters.u0 < parameters.max)
                      THEN parameters.u0 + interface.Insignal
                      ELSE parameters.max
              ELSE 0 ;
    };
  :};

CLASS Sensor
   (* class = "sensor"; category = "simple" *)
    "sensor"
  SPECIALISES PhysicalDAEModelType :=
    {:
    interface <-
    {
    OBJ Inflow (* terminal = "in" *) "inflow" :
          inWWTPTerminal := {: causality <- "CIN" :};
    OBJ Outflow (* terminal = "out_1" *) "outflow" :
          outWWTPTerminal := {: causality <- "COUT" :};
    OBJ Outsignal (* terminal = "out_2" *) "Sensor measured oxygen output" :
          Signal := {: causality <- "COUT"  :};
    };

    equations <-
    {
      {FOREACH comp_index IN {1 .. NrOfComponents}:
        interface.Outflow[comp_index] = interface.Inflow[comp_index];};

      interface.Outsignal = interface.Inflow[S_O];
```

```
    };
  :};

CLASS Subtractor
  (* class = "subtractor"; category = "simple" *)
  "subtract_In1_In2"
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
    {
     OBJ Insignal1  (* terminal = "in_1" *) " + " :
               Signal := {: causality <- "CIN" :};
     OBJ Insignal2  (* terminal = "in_2" *) " - " :
               Signal := {: causality <- "CIN" :};
     OBJ Outsignal (* terminal = "out" *)  "Insignal1 - Insignal2" :
               Signal := {: causality <- "COUT" :};

    };
  equations <-
    {
      interface.Outsignal = interface.Insignal1 - interface.Insignal2 ;
    };
  :};


CLASS Subtractor2
  (* class = "subtractor"; category = "simple" *)
  "subtract_In1_In2"
  SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
    {
     OBJ Insignal1  (* terminal = "in_1" *)
            " +, value of the set_point " :
               Signal := {: causality <- "CIN" :};
     OBJ Insignal2  (* terminal = "in_2" *)
            " - " :
               Signal := {: causality <- "CIN" :};
     OBJ Outsignal (* terminal = "out" *)
            "Insignal1 - Insignal2" :
               Signal := {: causality <- "COUT" :};

    };
  equations <-
    {
      interface.Outsignal = interface.Insignal1 - interface.Insignal2 ;
    };
  :};


// PLANT

CLASS simpleWWTModel
   (* class = "activated_sludge_unit"; category = "simple" *)
   SPECIALISES PhysicalDAEModelType :=
  {:
    interface <-
    {
     OBJ Inflow (* terminal = "in_1" *) "inflow" :
          inWWTPTerminal := {: causality <- "CIN" :};
     OBJ Outflow (* terminal = "out" *) "outflow" :
           outWWTPTerminal := {: causality <- "COUT" :};
```

```
 OBJ Insignal (* terminal = "in_2" *)
     "Oxygen transfer coefficient, i.e Kla" :
      Signal := {: causality <- "CIN" :};
};

parameters <-
 {
 OBJ S_O_sat  "Oxygen saturation concentration"
  : PhysicalQuantityType := {: value <- 9.1 ;  unit <- "mg/l" :};
 OBJ Y_H      "Yield For Heterotrophic Biomass"
 : Yield := {:value <- 0.67:};
 OBJ f_P      "Fraction Of Biomass Converted To Inert Matter"
 : PhysicalQuantityType := {:value<- 0.08:};
 OBJ mu_H     "Maximum Specific Growth Rate For Heterotrophic Biomass"
  : GrowthRate := {:value <- 6.00:};
 OBJ K_SS     "Half-Saturation Coefficient For Heterotrophic Biomass"
  :  PhysicalQuantityType :={:value <- 20.00:};
 OBJ K_SO  "Oxygen Half-Saturation Coefficient For Heterotrophic Biomass"
  :  PhysicalQuantityType := {:value <- 0.2:};
 OBJ b_H      "Decay Coefficient For Heterotrophic Biomass"
  : PhysicalQuantityType := {:value <- 0.62:};
 };

 independent <-
 {
 OBJ t "independent variable time": Time ;
 };

 state <-
 {
 // flow variables: Q/V, S_O, S_S, X_BH
 OBJ C: PhysicalQuantityType[NrOfComponents;];

 // algebraic variables
 OBJ ro: PhysicalQuantityType;
 OBJ rg: PhysicalQuantityType;
 OBJ rd: PhysicalQuantityType;
 };

 equation <-
 {
  state.C[Q_V] = interface.Inflow[Q_V] ;

  DERIV(state.C[S_O],[independent.t]) =
    interface.Insignal*(parameters.S_O_sat - state.C[S_O])
    +  state.C[Q_V]*(interface.Inflow[S_O] - state.C[S_O])
    - state.ro ;

  DERIV(state.C[S_S],[independent.t]) =
     state.C[Q_V]*(interface.Inflow[S_S] - state.C[S_S])
    - state.rg/parameters.Y_H ;

  DERIV(state.C[X_BH],[independent.t]) =
     state.C[Q_V]*(interface.Inflow[X_BH] - state.C[X_BH])
    + state.rg - state.rd ;

  state.ro =
    (1 - parameters.Y_H)/parameters.Y_H *state.rg
    + (1 - parameters.f_P)*state.rd ;

  state.rg =
    parameters.mu_H*state.C[S_O]/(parameters.K_SO + state.C[S_O])
```

```
        *state.C[S_S]/(parameters.K_SS + state.C[S_S])
        *state.C[X_BH] ;

      state.rd =
        parameters.b_H*state.C[X_BH] ;

      {FOREACH comp_index IN {1 .. NrOfComponents}:
        interface.Outflow[comp_index] = state.C[comp_index];};
     };
   :};


// CONTROLLERS

CLASS Direct_controller
  (* class = "controller"; category = "simple" *)
  "controller"
  SPECIALISES
  PhysicalDAEModelType :=
  {:
    interface <-
    {
     OBJ Insignal (* terminal = "in" *)
           "Input Signal, i.e (set_point - variable to set to set_point) " :
               Signal := {: causality <- "CIN" :};
     OBJ Outsignal (* terminal = "out" *) "Manipulated variable" :
               Signal := {: causality <- "COUT" :};

    };
    parameters <-
    {
     // Controller parameters
     OBJ a2 : RealNumbers := {: value <- -0.1680159049 :};
     OBJ a3 : RealNumbers := {: value <- - 2.06412848 :};
     OBJ b1 : RealNumbers := {: value <- 865.0207436 :};
     OBJ b2 : RealNumbers := {: value <- 672.733843 :};
     OBJ b3 : RealNumbers := {: value <- 120.1695874 :};
     OBJ d  : RealNumbers := {: value <- 0.1119282746 :};
    };

    independent <-
    {
     OBJ t "independent variable time": Time
    };

    state <-
    {
     OBJ control_1: PhysicalQuantityType ;
     OBJ control_2: PhysicalQuantityType ;
     OBJ control_3: PhysicalQuantityType ;
    };

    equations <-
    {
     DERIV(state.control_1, [independent.t]) = state.control_2 ;
     DERIV(state.control_2, [independent.t]) = state.control_3 ;
     DERIV(state.control_3, [independent.t]) = interface.Insignal
       + parameters.a2*state.control_2 + parameters.a3* state.control_3 ;

     interface.Outsignal =
       parameters.b1*state.control_1 + parameters.b2*state.control_2
       + parameters.b3*state.control_3 + parameters.d*interface.Insignal ;
```

```
    };
  :};

CLASS Integral_controller
  (* class = "controller"; category = "simple" *)
  "controller"
  SPECIALISES
  PhysicalDAEModelType :=
  {:
    interface <-
    {
     OBJ Insignal (* terminal = "in" *)
            "Input Signal, i.e variable to set to set_point " :
               Signal := {: causality <- "CIN" :};
     OBJ Outsignal (* terminal = "out" *) "Manipulated variable" :
               Signal := {: causality <- "COUT" :};

    };

    independent <-
    {
     OBJ t "independent variable time": Time
    };

    state <-
    {
     OBJ control : PhysicalQuantityType ;
    };

    equations <-
    {
     DERIV(state.control, [independent.t]) =
          interface.Insignal ;

     interface.Outsignal = state.control ;
    };
  :};

CLASS PID_controller
  (* class = "controller"; category = "simple" *)
  "controller"
  SPECIALISES
  PhysicalDAEModelType :=
  {:
    interface <-
    {
     OBJ Insignal (* terminal = "in" *)
            "Input Signal, i.e  (set_point - variable to set to set_point) " :
               Signal := {: causality <- "CIN" :};
     OBJ Outsignal (* terminal = "out" *) "Manipulated variable" :
               Signal := {: causality <- "COUT" :};

    };
    parameters <-
    {
     // PID parameters
     OBJ kp : PhysicalQuantityType := {: value <- 11.31977725 :};
     OBJ Ti : PhysicalQuantityType := {: value <- 0.5062161847 :};
     OBJ Td : PhysicalQuantityType := {: value <- 0.2531080923 :};
    };

    independent <-
```

```
    {
     OBJ t "independent variable time": Time
    };

    state <-
    {
     OBJ control_1: PhysicalQuantityType ;
     OBJ control_2: PhysicalQuantityType ;
     OBJ control_3: PhysicalQuantityType ;
    };

    equations <-
    {
     DERIV(state.control_1, [independent.t]) = state.control_2 ;
     DERIV(state.control_2, [independent.t]) =
       -(10/parameters.Td)*state.control_2 + interface.Insignal ;

     interface.Outsignal =
       + (10*parameters.kp/(parameters.Ti*parameters.Td))*state.control_1
       + (-10*parameters.kp/parameters.Td
       + (parameters.Td*parameters.kp
       + 10*parameters.kp*parameters.Ti)/(parameters.Td*parameters.Ti))
       *state.control_2
       + parameters.kp*interface.Insignal ;
    };
  :};


CLASS IMC_controller
  (* class = "controller"; category = "simple" *)
  "controller"
  SPECIALISES
  PhysicalDAEModelType :=
  {:
    interface <-
    {
     OBJ Insignal (* terminal = "in" *)
          "Input Signal, i.e (set_point - ( variable to set to set_point-model output) " :
              Signal := {: causality <- "CIN" :};
     OBJ Outsignal (* terminal = "out" *) "Manipulated variable" :
              Signal := {: causality <- "COUT" :};
    };

    parameters <-
    {
     OBJ a " filter parameter":  PhysicalQuantityType := {: value <- 0.1 :};
    };

    independent <-
    {
     OBJ t "independent variable time": Time
    };

    state <-
    {
     OBJ control_1: PhysicalQuantityType ;
     OBJ control_2: PhysicalQuantityType ;
     OBJ control_3: PhysicalQuantityType ;
    };

    equations <-
    {
```

```
     DERIV(state.control_1, [independent.t]) = state.control_2 ;
     DERIV(state.control_2, [independent.t]) = state.control_3 ;
     DERIV(state.control_3, [independent.t]) =
        interface.Insignal
        - 0.1680159049/parameters.a * state.control_1
        - 1/parameters.a * state.control_2
            *(0.1680159049*parameters.a + 2.064128479)
        - 1/parameters.a * state.control_3
            *(1 + 2.064128479*parameters.a) ;


     interface.Outsignal =  1/parameters.a *
          ( (865.0207428 -   0.01880573034/parameters.a) * state.control_1
          + (672.7526482
             -(0.01880573034*parameters.a + 0.2310343392)/parameters.a)
                *state.control_2
          + (120.4006217
             -(0.2310343392*parameters.a + 0.1119282746)/parameters.a)
                *state.control_3
          + 0.1119282746*interface.Insignal) ;
    };
  :};

CLASS IMC_model
 (* class = "controller"; category = "simple" *)
  "controller"
  SPECIALISES
  PhysicalDAEModelType :=
  {:
    interface <-
    {
     OBJ Insignal (* terminal = "in" *)
            "Input Signal, i.e. Control OutSignal " :
               Signal := {: causality <- "CIN" :};
     OBJ Outsignal (* terminal = "out" *)
            "Model of the observable variable" :
               Signal := {: causality <- "COUT" :};

    };

    independent <-
    {
     OBJ t "independent variable time": Time
    };

    state <-
    {
     OBJ so : PhysicalQuantityType ;
     OBJ ss : PhysicalQuantityType ;
     OBJ xbh : PhysicalQuantityType ;
    };

    equations <-
    {
    DERIV(state.so, [independent.t]) =
       -1073.630304*state.so - 0.02116201945*state.ss
       - 1.860847195*state.xbh + 8.934293000*interface.Insignal ;

    DERIV(state.ss, [independent.t]) =
       -3095.849407*state.so - 2.064127331*state.ss
       - 3.910446047*state.xbh ;
```

```
    DERIV(state.xbh, [independent.t]) =
       2074.219103*state.so + 0.04296531223*state.ss
       - 0.000001147972827*state.xbh ;

     interface.Outsignal =  state.so ;
    };
  :};


// End of LibAeration.msl
```

Bibliography

[1] J.D. Bailey and D.F. Ollis. *Biochemical Engineering Fundamentals*. McGraw-Hill, 1986.

[2] S.P. Banks. *Control Systems Engineering*. Prentice-Hall, 1986.

[3] Filip Claeys. West++: Modelling environment, hierarchical graph editor server, 1998.

[4] Henze, Harremoes, Jansen, Arvin. *Wastewater Treatment*. Springer, 1995.

[5] J.H. Davenport, Y. Siret, E. Tournier. *Computer Algebra*. Academic Press, 1993.

[6] Prof. Ing. Antonino Liberatore, editor. *Manuale Cremonese di MECCANICA, ELETTROTECNICA, ELETTRONICA*, volume 1. Edizioni Cremonese, 1992.

[7] M. Henze, C.P.L. Grady, W. Gujer, G.v.R. Marais and T. Matsuo. Activated sludge model no. 1, 1987.

[8] Manfred Morari, Evanghelos Zafiriou. *Robust Process Control*. Prentice-Hall International, 1989.

[9] Giovanni Marro. *Controlli Automatici*. Zanichelli, 1996.

[10] The MATH WORKS Inc. *MATLAB Control System Toolbox, User's Guide*, 1996.

[11] The MuPAD Group. *MuPAD User's Manual*, 1996.

[12] H. Vangheluwe and G. Vansteenkiste. Computer-aided modelling of complex systems. In *Proceedings of the ADIUS Sixteenth Annual Conference*. ADI, 3800 Stone School Rd., Ann Arbor, Michigan 48108-2499 USA, 1995.

[13] Hans Vangheluwe. West++: Modelling environment, msl-user, 1998.

[14] S.M. Walas. *Modelling with Differential Equations in Chemical Engineering*. Butterworth-Heinemann, 1991.