

**SCK•CEN**  
Department of Reactor Physics & Myrrha

**Internship Report**

**Optimizing Nuclear Reactor Control Using Soft  
Computing Techniques**

Internship on Nuclear Reactor Control With Soft Computing Techniques  
at the Belgian Nuclear Research Center SCK•CEN  
by

**Jorg Onno Entzinger**

UT/WB-2004.s9905103-24.11.2004

Enschede 2004



**THE BELGIAN NUCLEAR RESEARCH CENTRE**

SCK•CEN  
Department of Reactor Physics & Myrrha  
Boeretang 200  
B-2400 Mol  
Belgium

University of Twente  
Department of Mechanical Automation  
PO-Box Postbus 217  
NL-7500 AE Enschede  
The Netherlands



# Optimizing Nuclear Reactor Control Using Soft Computing Techniques

Internship on Nuclear Reactor Control With Soft Computing Techniques  
at SCK•CEN  
by

**Jorg Onno Entzinger**

UT/WB-2004.s9905103-24.11.2004

Supervisor: **D. Ruan**  
SCK•CEN

Examinator: **J.B. Jonker**  
University of Twente

Enschede, 24 November 2004



# Abstract

This report presents the results of an investigation of different soft computing techniques for application in nuclear reactor control. All research and tests have been carried out using a computer simulation of a demonstration model, consisting of an emptying water tank which is refilled by two controlled flows. This demo provides a safe and representative test bed, because despite of its simplicity it provides a highly non-linear control problem.

Starting with a standard fuzzy logic controller, different techniques have been applied to improve the performance and robustness and to generalize the controller design process. Rule base adaptation using only two simple guiding rules and membership optimization using genetic algorithms have resulted in a promising method to design high-performance controllers for industrial applications.

To enable off-line adaptation and optimization, which would be needed due to safety regulations, plant modeling using neural networks has been investigated. Although this technique matches best with the characteristics of the other methods applied (problem independence, no mathematical formulation is needed, high robustness), this appears to be very difficult because of the inaccuracy inherent to neural networks.



# Acknowledgements

I would like to thank my supervisor Da Ruan very much for sharing all his knowledge and guiding me with cheerfulness and enthusiasm, not only when we were talking about my assignment, but also when it came to personal development. I appreciated the freedom he gave me to find my own way through the matter and the support he gave me when decisions had to be made. He and the other students at the SCK•CEN made my internship a period I look back on happily.



# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction</b>                               | <b>1</b>  |
| 1.1      | Control Problem . . . . .                         | 1         |
| 1.1.1    | Nuclear Reactor Control . . . . .                 | 1         |
| 1.1.2    | Control of a Demonstration Model . . . . .        | 3         |
| 1.2      | Directions of exploration . . . . .               | 4         |
| <b>2</b> | <b>Soft Computing</b>                             | <b>7</b>  |
| 2.1      | Fuzzy Logic (FL) . . . . .                        | 8         |
| 2.1.1    | Overview . . . . .                                | 8         |
| 2.1.2    | Building a Fuzzy Logic Controller (FLC) . . . . . | 8         |
| 2.1.3    | History . . . . .                                 | 11        |
| 2.1.4    | References . . . . .                              | 11        |
| 2.2      | Neural Networks (NNs) . . . . .                   | 11        |
| 2.2.1    | Overview . . . . .                                | 11        |
| 2.2.2    | History . . . . .                                 | 14        |
| 2.2.3    | References . . . . .                              | 14        |
| 2.3      | Genetic Algorithms (GAs) . . . . .                | 14        |
| 2.3.1    | Overview . . . . .                                | 14        |
| 2.3.2    | History . . . . .                                 | 15        |
| 2.3.3    | References . . . . .                              | 15        |
| <b>3</b> | <b>Fuzzy Logic Controllers</b>                    | <b>17</b> |
| 3.1      | FLC with a Static Rule Base . . . . .             | 18        |
| 3.2      | FLC with an Adaptive Rule Base . . . . .          | 20        |
| 3.3      | Input scaling . . . . .                           | 22        |
| 3.4      | 5 versus 7 Membership Functions . . . . .         | 24        |
| 3.5      | Performance with Changing Flow Rates . . . . .    | 26        |
| <b>4</b> | <b>Application of Neural Networks</b>             | <b>29</b> |
| 4.1      | Neural Network Plant Models . . . . .             | 29        |
| 4.1.1    | Background . . . . .                              | 29        |
| 4.1.2    | Neural Networks . . . . .                         | 31        |
| 4.2      | Neuro (Fuzzy) Controllers . . . . .               | 34        |

|          |   |           |
|----------|---|-----------|
| <b>5</b> | <b>Genetic Fuzzy Optimization</b>             | <b>39</b> |
| 5.1      | Genetic Fuzzy Optimization . . . . .          | 39        |
| 5.1.1    | Genetic Parametrization . . . . .             | 40        |
| 5.1.2    | Fitness Function . . . . .                    | 41        |
| 5.1.3    | Results . . . . .                             | 41        |
| 5.2      | Genetic Adaptive Fuzzy Optimization . . . . . | 44        |
| <b>6</b> | <b>Conclusion</b>                             | <b>47</b> |
| <b>A</b> | <b>Implementation</b>                         | <b>53</b> |
| A.1      | Fuzzy Inference System (FIS) . . . . .        | 53        |
| A.2      | Water Tank . . . . .                          | 53        |
| A.3      | Error Messages . . . . .                      | 58        |
| <b>B</b> | <b>Matlab Code</b>                            | <b>60</b> |
| B.1      | General Code . . . . .                        | 60        |
| B.1.1    | initFIS.m . . . . .                           | 60        |
| B.1.2    | makeFIS.m . . . . .                           | 61        |
| B.1.3    | addStaticRules.m . . . . .                    | 62        |
| B.1.4    | addAdaptiveRules.m . . . . .                  | 63        |
| B.1.5    | vlinspace.m . . . . .                         | 65        |
| B.1.6    | evalStatic.m . . . . .                        | 65        |
| B.1.7    | evalAdapt.m . . . . .                         | 65        |
| B.2      | NN Code . . . . .                             | 67        |
| B.2.1    | evalNN.m . . . . .                            | 67        |
| B.2.2    | makeNN.m . . . . .                            | 68        |
| B.2.3    | compareNN.m . . . . .                         | 72        |
| B.2.4    | compareNNSim.m . . . . .                      | 74        |
| B.3      | GA Code . . . . .                             | 74        |
| B.3.1    | optimizeLong.m . . . . .                      | 74        |
| B.3.2    | fisEval3LongNew.m . . . . .                   | 77        |
| B.3.3    | editFIS.m . . . . .                           | 80        |

# Chapter 1

## Introduction

This report covers my work at the Belgian nuclear research institute SCK•CEN in May and June 2004. The official assignment was: *Apply neural networks and/or genetic algorithms for adaptive fuzzy rule base optimization*. The purpose of this project is to explore new possibilities for a better system to control the power output of a nuclear reactor.

Safety regulations for nuclear reactor control are very strict, which makes it much more difficult to implement new techniques. One such technique is fuzzy logic (§2.1), which provides very desirable advantages over classical, like robustness, adaptation and the capability to include human experience into the controller. In the end however, basic fuzzy logic controllers were implemented in some nuclear reactors like the Massachusetts Institute of Technology (MIT) research reactor [1] in 1988 and the first Belgian reactor (BR1) in 1998, though only on a temporal basis.

The research done within the framework of my internship is a continuation of earlier research on adaptive fuzzy logic controllers for nuclear reactors at the SCK•CEN ([2], [3], [4]). It will not immediately lead to a new controller to be implemented in a nuclear reactor. It will only provide some information to support future decisions on the directions to go for optimization of reactor control operation. This means that during the exploration process there is more freedom to try things that do not meet all current requirements on nuclear reactor control.

For readers unacquainted with soft computing an overview of the different methods used in this report (fuzzy logic, neural networks and genetic algorithms) is presented in Chapter 2.

### 1.1 Control Problem

#### 1.1.1 Nuclear Reactor Control

The control problem faced is that of the power output of a nuclear reactor. The current controller only maintains a steady-state power level and is not used for

setpoint changes. Of course also these setpoint changes should be carried out by a controller in future implementations.

Nuclear reactors have three key elements:

- radioactive fuel
- a moderator
- control rods

When the fuel is bombarded by a free, low-energy neutron, the atoms split into two major fission fragments and releasing high-energy neutrons as well as energy in the form of heat. These new, free, but high-energy neutrons however, are less likely to trigger other atoms to split. Therefore a moderator such as carbon (graphite) or hydrogen (in the form of water) is needed, so the high-energy neutrons lose their energy by colliding into and bouncing off of the moderator atoms. Now neutrons are slowed down they will trigger new atoms to split, causing a chain reaction (see Figure 1.1).

To control the chain reaction, control rods are inserted into or withdrawn from the reactor core. These rods are made of a material that absorbs the neutrons such as cadmium or boron. This way free neutrons are taken out of the chain reaction and thus the process is tempered. Of course for a steady state process it is important to have one free neutron left from each fission. When the power output must raise, control rods are temporarily withdrawn a little to keep slightly more free neutrons, increasing the number of fissions and thus the power level. When the power is near the desired level, the control rod are inserted again to have a new steady state, keeping one free neutron from each fission.

There are three types of control rods:

- shim rods (also C-rods, for coarse control)
- regulating rods (also A-rods, for fine adjustments)
- safety rods (for very fast shutdown)

For a normal controller only the first two types are interesting, because the safety rods will only be controlled manually or by a supervisory controller. Normally only the regulation rods are active. They level out small changes in the power output caused for instance by a change in reactivity due to a raise in temperature. When the regulation rods are almost fully inserted or withdrawn, the shim rods are moved slightly, allowing the regulation rods to move back to a central position. Further the shim rods are only used at startup, shutdown and setpoint changes.

Currently the BR1 nuclear reactor at the SCK•CEN is controlled by a conventional simple on-off controller which only controls the regulation rods. The shim rods have to be moved manually. An early study of fuzzy logic control for the BR1 reactor ([2], [3]) resulted in a 1-year operation of such a controller, which actually

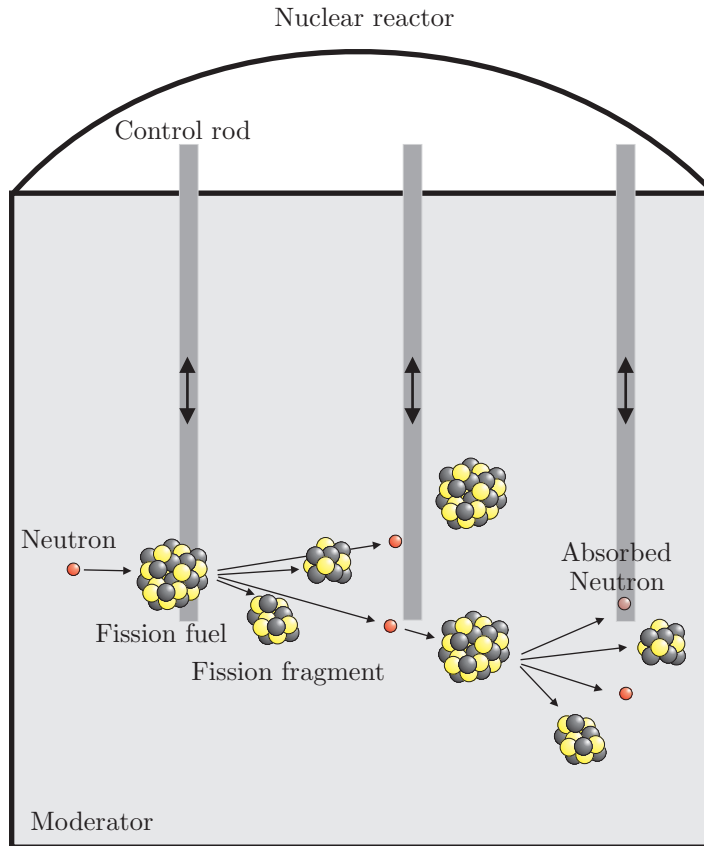


Figure 1.1: Working principle of a nuclear fission reactor.

did combine the shim rod and regulation rod control, however only in steady state situations.

Physically all control rods are the same, they only differ in number, position in the reactor and function. For instance the shim rods are positioned in the center of the reactor core where a lot of neutrons pass and their effect is big, whereas the regulation rods are placed more to the sides.

### 1.1.2 Control of a Demonstration Model

Because it is difficult and time consuming to make a well resembling mathematical model of a nuclear reactor and moreover, the set of nonlinear differential equations would be hard to solve, newly developed controllers are tested on a demo model ([4] and Ch. 4 of [5]). This model consists of a water tank which empties constantly through a small hole in the bottom and is filled simultaneously by two water flows

as depicted in Figure 1.2. These filling flows are controlled by valves, so the water level in the main tank can be regulated. In this model the water level in the main

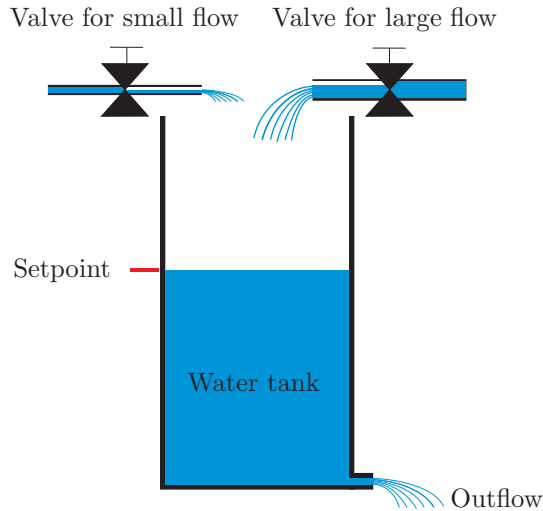


Figure 1.2: Water tank demo model.

tank resembles the power output of the nuclear reactor. The speed of opening or closing the valves reflects the speed of inserting or withdrawing the control rods. One of the filling flows is made significantly larger than the other to copy the difference between shim rods and control rods.

Though this water tank demo model is much simpler than the nuclear reactor, it is still a non-linear system which is difficult to control. The demo therefore is a good and representative testbed for the controllers investigated.

For the simulations described in this report, a slightly changed version of the tank and valves from the MATLAB fuzzy logic toolbox ‘water tank demo’ were used. For more information please refer to Appendix A.2.

## 1.2 Directions of exploration

For nuclear power plants it is important to respond quickly on a change in power demand. At the same time overshoot should be minimal both for safety reasons and because it is a costly waste of energy. With these problems we find again the classical control problem: the controller should be both fast and accurate.

In this report several options are investigated to improve a static fuzzy logic controller for speed, stability and accuracy. In this investigation soft computing techniques (Chapter 2) are used, mainly because of their robustness, their learning capabilities and their high level of problem independence. Options investigated are:

- Adaptive rule generation (Chapter 3)
- Number of membership functions (§3.4)
- Fuzzy Neural Networks (§4.2)
- Use of Neural Networks for plant simulation (§4.1)
- Optimization of membership functions by Genetic Algorithms (Chapter 5)

A setup with an adaptive fuzzy rule base was already present from earlier research ([4] and ch. 4 of [5]). Most tests are performed with both a static rule base and the adaptive version.

For implementation the MATLAB [6] programming language has been chosen, because of its flexibility, the large amount of predefined algorithms and the easy use. The drawback of this choice is maybe the limited possibilities MATLAB offers for neurofuzzy implementations, however this could be solved by using toolboxes written by other MATLAB users.

It must be stressed that only a few possibilities are investigated. Some other options like different rule base adaptation mechanisms, membership function adaptation, plant identification techniques and advanced neuro-fuzzy systems might be interesting. However time did not allow a more extensive research. Also non-fuzzy control and combinations of fuzzy logic with classical control have not been investigated, but are very well possible.

Because soft computing solutions do not require a lot of mathematical modeling and also few mathematical principles are used in the algorithms themselves, it is often difficult to obtain hard evidence for properties like robustness, efficiency, etcetera. Most conclusions therefore will be drawn by extracting the general trends of multiple runs with slightly different (mostly randomized) inputs.

Though stability of fuzzy controllers can be proved mathematically, this is more a tough mathematical exercise than it is of use for a practical controller. Because of their transparency and human like behavior, fuzzy controllers can often easily be checked by just having a close look at their rule base.



## Chapter 2

# Soft Computing

This chapter introduces the soft computing techniques dealt with in this research. Readers who are familiar with soft computing may want to skip this chapter.

Soft computing is a generic term for a group of methods that is tolerant of imprecision, uncertainty, partial truth, and approximation. This tolerance results in a high robustness, which makes these methods very interesting in a broad range of application fields. The principal constituents of soft computing are:

- Fuzzy Logic (rule based reasoning and decision making, see also §2.1)
- Neural Computing (learning algorithm for approximation and classification, see also §2.2)
- Evolutionary Computing (optimization, see also §2.3)

The different segments of soft computing mentioned above have very distinct and complementary properties. Often these methods are combined or so called ‘hybrid’ methods like *Neuro Fuzzy* and *Fuzzy Evolutionary Computing* are applied to obtain a system with the advantages of all methods.

Soft computing algorithms only need very basic knowledge of the system they act upon. The fact that soft computing focuses on the behavior of these systems instead of on the way they work, makes a mathematical formulation of the systems unnecessary. Therefore soft computing can be a very powerful tool for a wide range of problems which are hard to solve with conventional algorithms because of their vagueness, uncertainty or complexity.

The tolerant and imprecise way of dealing with problems the different soft computing techniques share can also often be found in nature, where we rarely find true ‘mathematical precision’ but often find very good solutions. Therefore it is not surprising that most soft computing techniques are strongly inspired nature. This resemblance also makes soft computing the basis for many so called ‘artificial intelligent’ systems.

For an overview of soft computing applications, please refer to [7], to [5] for applications in nuclear engineering and to [8] for an extensive overview of appli-

cations of fuzzy logic. The standard work on fuzzy logic, neuro fuzzy and fuzzy genetic algorithms for control applications is ‘Modeling and Control’ [9] from the Fuzzy Systems series.

## 2.1 Fuzzy Logic (FL)

### 2.1.1 Overview

A Fuzzy Logic system is an inference system that uses if-then rules to draw a conclusion for a set of presented inputs. It allows computers to use more than just true or false, to work more easily with phrases such as ‘fairly’, ‘rarely’, or ‘almost’. Fuzzy logic is not called fuzzy because the reasoning logic used would be fuzzy, but because the variables used do not take crisp values but linguistic values like ‘big’, ‘small’ and ‘near ...’, which are less exact. This makes FL systems easy to work with in daily life, where we do not say the door is 47.2% opened but we say it is ‘almost half’ opened and to close it we don’t talk about giving it an impulse of  $0.11N \cdot s$  but about giving it a ‘little’ push.

The strength of Fuzzy Logic Controllers (FLC) is that they speak the same language as we do. Therefore expert knowledge can easily be put into a FLC, telling the controller how to react to different inputs. This means no exact model is needed of the system to be controlled, as long as there is a global understanding of the influence of the control variables on the system output. In the case of closing the door this means we do not need to know all about the hinges, inertia and friction, we only need our expert knowledge that a soft push (in the right direction) will close a half opened door and a ‘slightly bigger’ push is needed for a fully opened door.

Another important property of FLCs is their robustness. Because transitions between input states are smooth, outputs will generally follow this smoothness. Where in classical control an imprecisely modeled system can easily cause the controller to become unstable, FLCs are to a large extent indifferent to such impreciseness or uncertainties.

### 2.1.2 Building a Fuzzy Logic Controller (FLC)

In this example a simple Fuzzy Logic central heater controller will be set up. First we define the problem in terms of variables:

- input: temperature error = current temperature - desired temperature
- output: change in heating power

First we have to ‘fuzzyfy’ these variables, which means that we must define the (linguistic) values these variables can take. For the input we choose:

- NL = negative large (when it is much too cold)
- NS = negative small (when it is a little too cold)

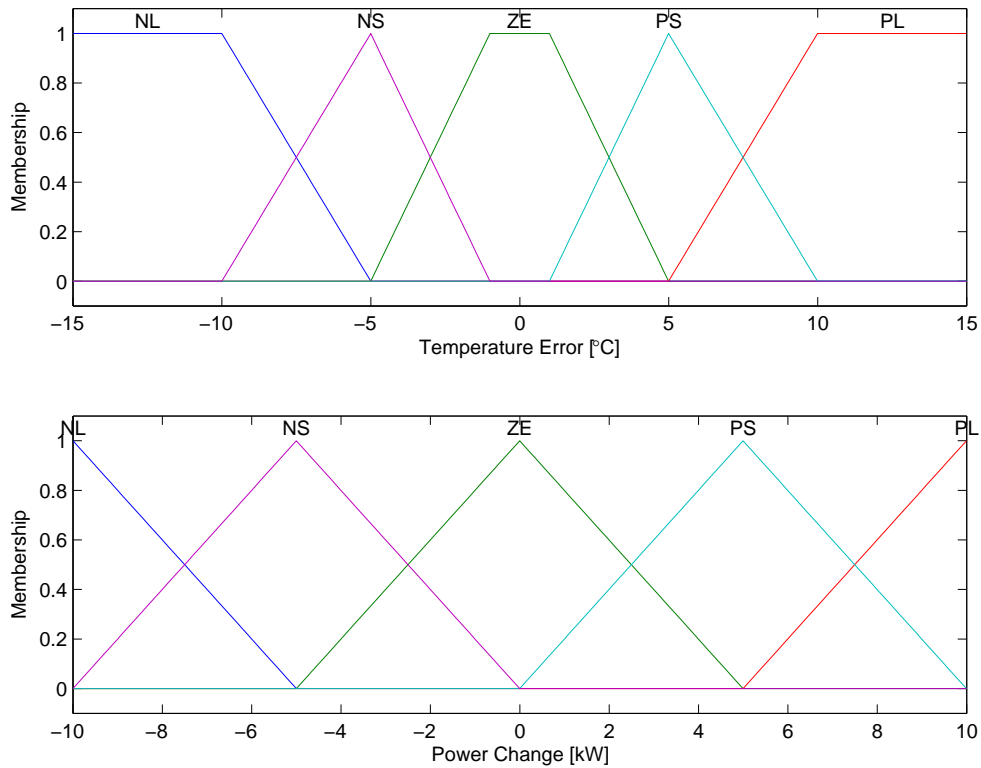


Figure 2.1: Membership functions for the input and output of a fuzzy controller.

- ZE = zero (when the temperature is ok)
- PS = positive small (when it is a little too hot)
- PL = positive large (when it is much too hot)

For the change in heating power we can use the same values:

- NL = negative large (to lower the power significantly)
- NS = negative small (to lower the power a bit)
- ZE = zero (to keep the power like it is)
- PS = positive small (to raise the power a bit)
- PL = positive large (to raise the power significantly)

We can now draw a picture of our input and output space (Figure 2.1). In this figure we see the so called membership functions for each value. These functions, which are often chosen triangular or trapezoidal, determine to which degree a certain crisp value on the x-axis belongs to each fuzzy value (i.e. NL, NS, ZE, PS

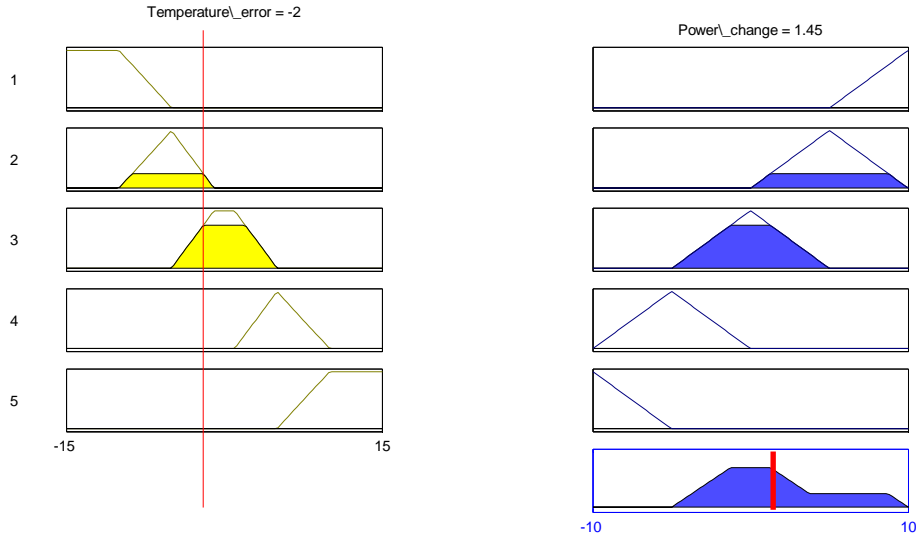


Figure 2.2: Activation of membership functions and the application of rules.

and PL). The positions and shapes of these membership functions can be chosen freely by the controller designer. In this case we see that a temperature error of 5 degrees belongs for 100% to the fuzzy value PS and an error of -7.5 degrees is 50% NL and 50% NS.

To make the controller actually work, we need a so called rule base. This rule base is a set of if-then rules connecting the inputs to the outputs. In this case the rules are quite simple:

- If  $TemperatureError = NL$  then  $PowerChange = PL$
- If  $TemperatureError = NS$  then  $PowerChange = PS$
- If  $TemperatureError = ZE$  then  $PowerChange = ZE$
- If  $TemperatureError = PS$  then  $PowerChange = NS$
- If  $TemperatureError = PL$  then  $PowerChange = NL$

From Figure 2.2 we see what happens inside the FLC when it is  $2^{\circ}C$  too cold in the room. The crisp input value  $-2$  is translated to partial memberships to the linguistic values NS and ZE. This means both rules (the rule for NS and the one for ZE) are activated to some extent, which results in the activation of the corresponding linguistic output values. These values are summed (as can be seen at the bottom right) and a 'center of mass' determines the crisp output value. The power of the heater will increase with  $1.45kW$ .

### 2.1.3 History

L. A. Zadeh [10] in 1965 first introduced fuzzy set theory to deal with imprecise objects. For a long time most engineers opposed to this new idea, until some 10 years later, when E. H. Mamdani [11] successfully applied fuzzy logic algorithms for the control of a small laboratory steam engine. After then, many applications of fuzzy control are reported in a wide range of engineering fields varying from decision making to pattern recognition and a variety of control systems [8]. Here we will focus on fuzzy logic control (FLC).

### 2.1.4 References

A very complete description of fuzzy logic methods can be found in [9]. An easy reading explanation with practical applications of fuzzy logic is given by von Altrock [12]. Also the MATLAB [6] fuzzy logic toolbox manual and demos can be very helpful.

## 2.2 Neural Networks (NNs)

### 2.2.1 Overview

Neural networks very much resemble the human brain, both in functioning and in structure. A NN is a set of interconnected neurons, has learning capabilities and can be used for classification problems and function mapping. There are a lot of different types of artificial NNs, each with their own pros and contras. When we do not consider NNs used for classification problems, there are roughly two types of networks: Feed Forward - Back Propagation (FF-BP) networks and Radial Basis (RB) networks. The main ideas of these types are described below.

The structure of a standard NN is depicted in Figure 2.3. The main elements of a neural network are:

- neurons
- weights
- biases

The neurons are often grouped in layers, in this case an input layer, one so called hidden layer and an output layer. In most networks only neurons in successive layers are interconnected, this is always the case for the feed-forward networks, which explains the name. The interconnection means that a neuron output value is presented as an input to the next neuron. Each neuron then calculates its output value based on its inputs.

Normal neurons work as depicted in Figure 2.4(a): the neuron's inputs are multiplied by the weights and then, together with the bias value, they are summed.

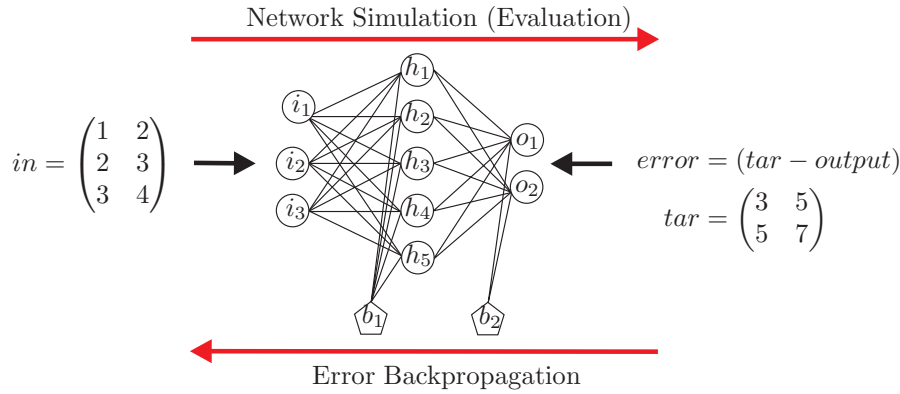


Figure 2.3: Typical shape of an artificial neural network;  $i$  = input layer neuron,  $h$  = hidden layer neuron,  $o$  = output layer neuron,  $b$  = bias.

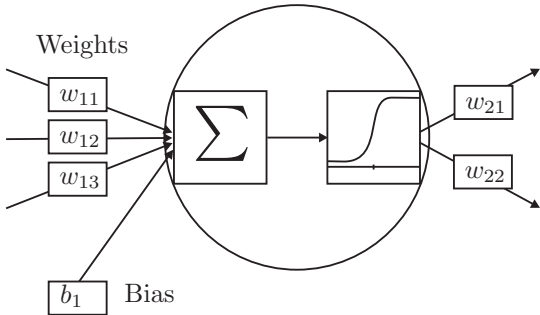
The value thus obtained is the input for a (smoothened) step function which determines the neuron's value. This value is then presented to all neurons in the next layer.

The hidden layer neurons in a RB network work a little different, as can be seen in Figure 2.4(b): here the neurons calculate the distance between the input values vector and the weights vector. This distance is then the input of a Gaussian function which determines the neuron's value.

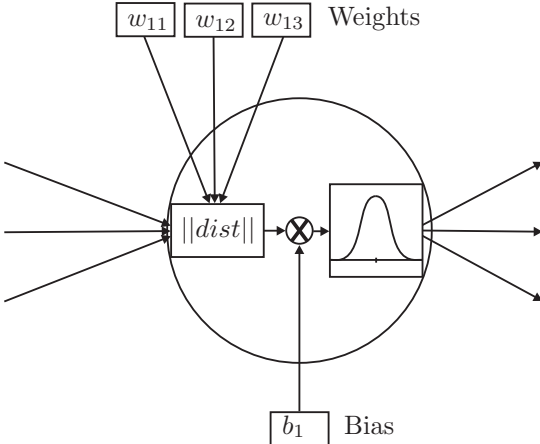
Before the NN can be evaluated, it must be trained, i.e. it has to 'learn' which values the weights and biases must take to reproduce certain target values when presented specific inputs. For the FF-BP networks this means that the inputs are fed through the network and the contribution of each connection weight and bias value is determined by passing the resulting errors through the network in a backward manner. The errors are then minimized by an iterative scheme such as Levenberg-Marquardt or Bayesian Regularization.

For the RB networks the weight and bias values of the hidden layer neurons are actually just 'set', because the weights of a certain neuron match the inputs of a certain training data set, the bias is a fixed value that determines the width of the Gaussian function. The weights and biases of the output layer neurons are calculated by solving a simple linear system. This means that RB NNs in general have as many neurons as there are training samples and that training is very fast because there is no iterative scheme. RB NNs however can also be made by just starting with one neuron and adding neurons one by one until a performance check on all data proves the network accurate enough. This will of course take more time for training, but networks with less neurons will evaluate considerably faster.

The major disadvantage of NNs is that they (like all soft computing techniques) do not provide exact answers. However with sufficient training simulation errors can be minimized, the output will always remain an approximation. On the other



(a) Feed-Forward Back Propagating



(b) Radial Basis

Figure 2.4: Different neuron implementations for FF-BP and RB networks.

hand NNs are (unlike lookup tables) well capable of processing inputs that were not trained, though this generalization is dependent of the problem complexity and the amount and quality of training data.

### 2.2.2 History

The first step toward artificial neural networks came in 1943 when Warren McCulloch, a neuro-physiologist, and a young mathematician, Walter Pitts, wrote a paper [13] on how neurons might work. They modeled a simple neural network with electrical circuits. In the late 1950's, Rosenblatt's work [14] resulted in a two-layer network, the perceptron, which was capable of learning certain classifications by adjusting connection weights.

With the improved computer technology of the 1980's neural network research took a high flight. The first major development was John Hopfield's paper [15] on what is now known as 'hopfield networks' in 1982. The other came a few years later, with the independent 'discoveries' of 'backpropagation' by Le Cun, Parker and the group of Rumelhart, Hinton and Williams. Kohonen's work on 'Self-Organizing Maps' was also first published in the early 1980's. From then on NNs became widely used in for instance investment analysis, character recognition and process control.

### 2.2.3 References

Some engineering applications of neural networks can be found in [16], [17] and [18]. For more practical and implementation related information, see the MATLAB [6] neural network toolbox help and demos.

## 2.3 Genetic Algorithms (GAs)

### 2.3.1 Overview

Genetic algorithms are guided random search techniques that work analogous to biological evolution. Much GA terminology is derived from the biological counterpart, e.g. there are individuals, populations, genes and fitnesses. An individual represents an element in the solution space. The individual's genes determine its location (i.e. specify all parameter values) and its fitness indicates the performance of the individual with respect to the objective and the constraints. Individuals are grouped in populations.

Assuming that a recombination of genetic material from well performing individuals creates favorable offspring, a GA uses selection and variation to reach a solution to an optimization problem. Selection is applied by assigning a higher probability to the genetic material of favorable individuals of getting passed on to the next generation. Variation is accomplished by means of crossover and mutation, i.e. recombination of genetic material from selected individuals, respectively random changes in genes.

The algorithm starts by creating an initial population consisting of individuals with randomly assigned genes. The fitness of each individual is determined using an appropriate evaluation function. Selection, crossover and mutation are applied consecutively to generate a new population (the next generation). This process is repeated until convergence is reached, that is, until a certain solution dominates the population.

A GA can handle more general classes of functions than traditional mathematical programming search techniques. Whereas the latter use characteristics of the problem (e.g. gradients and continuity), GAs don't require such assumptions. They can handle also non-differentiable and discontinuous functions. Because of the use of mutation and the fact that GAs operate on a population instead of a single starting point, GAs are less likely to get stuck in a local optimum. However, increased randomness comes with decreased convergence speed, therefore GAs typically require more function evaluations than gradient based optimizers.

### 2.3.2 History

The basis for GAs of the form we know nowadays, was laid by John Holland's work [19] on adaptive systems from 1962-1975. Holland was the first to explicitly propose crossover and other recombination operators. Other evolution strategies developed around that time were more similar to hill-climbers than to genetic algorithms: there was neither a population nor crossover. One parent was mutated to produce one offspring and the better of the two was kept and became the parent for the next round of mutation ( [20] p.146).

Until the early 1980s, the research in GAs was mainly theoretical, but from then on there was an abundance of applications which spread across a large range of disciplines [21]. At first, these applications were mainly theoretical. However, the ongoing research and the exponential growth of computing power finally helped GAs to gain the interest of the commercial sector. The work of Kenneth De Jong showed that GAs could perform well on a wide variety of test functions, including noisy, discontinuous, and multimodal search landscapes.

### 2.3.3 References

For the research dealt with in this report, the GAOT toolbox [22] for MATLAB was used. A good starting point for literature on GAs is the work of Michalewicz [23]. Examples of some engineering applications can be found in [24], [25], [26] and [18]. A more general overview is given at The Talk.Origins Archive website by Adam Marczyk [27] and in the *The Hitch-Hiker's Guide to Evolutionary Computation* [28] which is the FAQ for the COMP.AI.GENETIC NEWSGROUP.



## Chapter 3

# Fuzzy Logic Controllers

All controllers are built to maintain the water level in the demo model (as described in §1.1.2 and Appendix A.2) at the desired level. The performance of the examined controller will be related to its response on time to time setpoint changes, which is a more distinct and more interesting criterion for future applications than small disturbances in a steady state.

The first step in this research exists of implementing the theory described in the papers on the earlier controllers for the water tank demo ([4] and ch. 4 of [5]). Because the original implementation was lost, a new implementation in MATLAB. A short description can be found here, a more detailed explanation is given in Appendix A

The controlled variables are the speed of opening or closing the valve for the large water inflow ( $VL$ ) and the speed of opening or closing the valve for the small water inflow ( $VS$ ). The controller inputs are the error in the water level (i.e. the current level minus the setpoint value) denoted by  $D$  and the changing rate of the water level (i.e. the time derivative of  $D$ ) denoted by  $DD$ .

The values the linguistic variables can take, which are also the names of the membership functions (MFs) are:

- NB - Negative Big
- NM - Negative Middle (exists only when 7 MFs are used)
- NS - Negative Small
- ZE - (almost) Zero
- PS - Positive Small
- PM - Positive Middle (exists only when 7 MFs are used)
- PB - Positive Big

A distinction will be made between the ‘static’ controller which has a fixed rule base and an adaptive controller which has a rule base that is updated iteratively. The adaptive rule base introduces great flexibility, not only because it makes the controller (almost) problem independent, but also because the controller can react better to unforeseen events. However, in practice it is almost impossible to get an adaptive controller to work on-line in a nuclear reactor, due to safety regulations.

### 3.1 FLC with a Static Rule Base

The static FLC has been implemented using the rule base in Table 3.1 and the membership functions shown in Figure 3.1. The rule base table should be read as following: the value *PS* in the upper right corner of the *VL* rule base means: ‘**If** *D = PB* **and** *DD = NB* **then** *VL = PS*’.

| VL   |    |    |    |    |    | VS   |    |    |    |    |    |
|--|----|----|----|----|----|--|----|----|----|----|----|
| $\begin{array}{c} D \\ \backslash \\ DD \end{array}$ | NB | NS | ZE | PS | PB | $\begin{array}{c} D \\ \backslash \\ DD \end{array}$ | NB | NS | ZE | PS | PB |
| NB   | PB | PB | PB | PS | PS | NB   | ZE | ZE | ZE | ZE | ZE |
| NS   | PB | PB | PS | PS | PS | NS   | ZE | ZE | NS | PS | NS |
| ZE   | PB | PS | ZE | NS | NS | ZE   | ZE | PS | ZE | PS | PS |
| PS   | PS | ZE | NB | NB | NB | PS   | ZE | NS | ZE | NB | NB |
| PB   | ZE | NB | NB | NB | NB | PB   | ZE | NB | ZE | NB | NB |

Table 3.1: Static rule base for VL and VS

The result is a working controller, as can be seen in Figure 3.2. However the controller response to a setpoint change is rather slow. The objective of this research is to find new methods to improve this response.

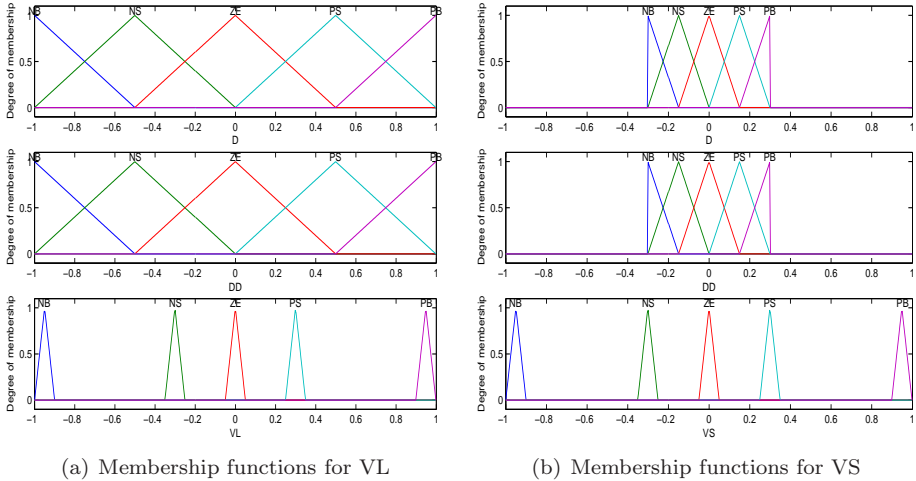


Figure 3.1: Input and output membership functions

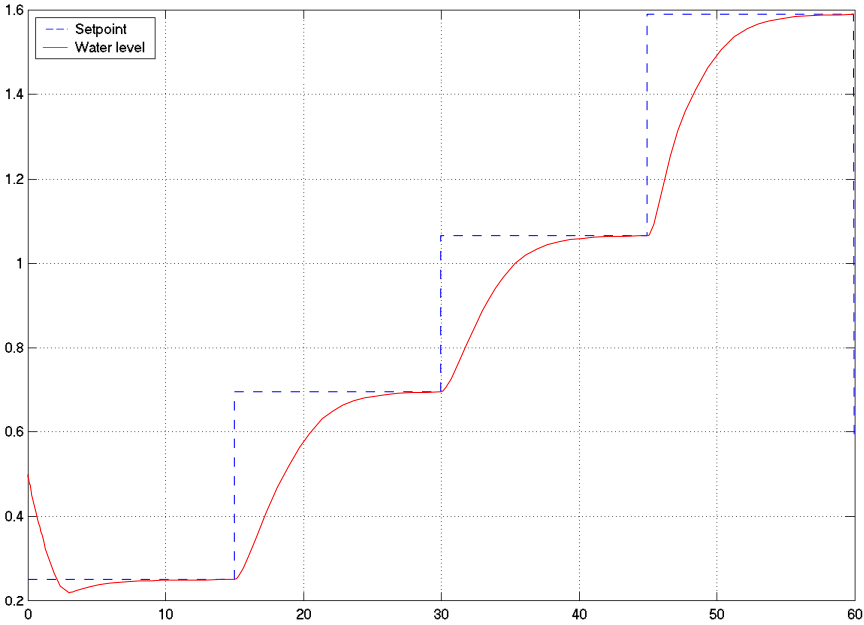


Figure 3.2: Water level control with the static controller.

### 3.2 FLC with an Adaptive Rule Base

The adaptive controller starts with the conclusion part of all rules set to Negative Big (see Table 3.2) and adapts its own rule base while controlling the system according to two simple rules:

- If the water level is too low and still dropping, open the valve faster (or close it slower)
- If the water level is too high, close the valve faster (or open it slower)

or in more technical terms:

- If  $D < 0$  and  $DD \leq 0$  then raise the conclusion part of the most triggered rule with 1
- If  $D > 0$  then lower the conclusion part of the most triggered rule with 1

Raising here means that next value out of  $\{NB, NS, ZE, PS, PB\}$  will be taken, i.e.  $NB$  becomes  $NS$ ,  $NS$  becomes  $ZE$  etcetera. The conclusion part for  $D = ZE$  &  $DD = ZE$  is always set to  $ZE$  to make sure the system will stabilize.

These rules are chosen in such a way that overshoot is minimized, thus restraining the speed of the controller. Addition of extra rules has been tried but did not provide a generally better solution: speed could only be gained at the expense of more overshoot.

| VL                |    |    |           |    |    | VS                |    |    |           |    |    |
|-------------------|----|----|-----------|----|----|-------------------|----|----|-----------|----|----|
| $D \backslash DD$ | NB | NS | ZE        | PS | PB | $D \backslash DD$ | NB | NS | ZE        | PS | PB |
| NB                | NB | NB | NB        | NB | NB | NB                | NB | NB | NB        | NB | NB |
| NS                | NB | NB | NB        | NB | NB | NS                | NB | NB | NB        | NB | NB |
| ZE                | NB | NB | <b>ZE</b> | NB | NB | ZE                | NB | NB | <b>ZE</b> | NB | NB |
| PS                | NB | NB | NB        | NB | NB | PS                | NB | NB | NB        | NB | NB |
| PB                | NB | NB | NB        | NB | NB | PB                | NB | NB | NB        | NB | NB |

Table 3.2: Initial rule base for VL and VS in an adaptive controller

The adaptive controller initializes itself very quick and it's response is often much faster than the response of the static controller (see Figure 3.3), which means that requests for more power can be met faster and there is less waste of energy when the power need decreases. Disadvantages are of course the overshoot (although it is only slight) and the (short) fluctuation when meeting a new setpoint value (for a solution see §3.3).

The adapted rule base is presented in Table 3.3, where the changed rules are printed bold and italic. It may seem that not many rules have adapted, but this is quite logical because the 4 rules in the lower left corner and the center rules will

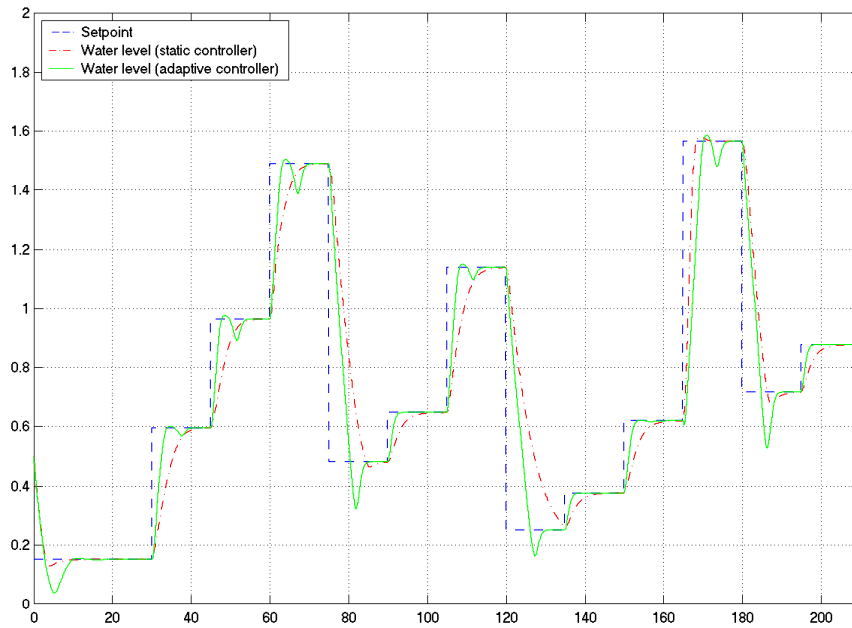


Figure 3.3: Comparison plot for the static and adaptive controller responses.

never change and the rules applying to a too high water level (i.e. the 2 most right columns) are supposed to make the water level decrease as soon as possible, which means they should have  $NB$  as a conclusion part. This means only the 8 upper left rules (center rule excluded) should change, which is exactly what happens. The upper rows do not change because of an improper scaling of the  $DD$  input variable.

A shortcoming of the rules stated previously is that they might not work fully when the controller needs to initialize itself with relatively small setpoint changes, as can be seen in Figure 3.4. The problem here is that the values of  $D$  and  $DD$  (Figure 3.5) are such that they both trigger the  $ZE$  membership function most for the  $VL$  rule base (Figure 3.1), this rule however is supposed to keep  $ZE$  as a conclusion. The controller has not been initialized properly yet (the  $VL$  rule base has not even changed from its initial state) causing the large valve to keep on closing slightly (because apart from the  $ZE$  membership functions, also some neighboring ones are triggered).

The small valve has opened fully in the mean time so the level is raising from 30s on, but with the increasing level the pressure and thus the outflow rate increase, establishing an equilibrium at a level of about 0.2m. The biggest problem here is that the controller cannot come out of its trap, because the rate is still positive (though very, very small) and therefore no adaption rules apply, even not when different  $VL$  membership functions are triggered.

| VL                |           |           |    |    |    | VS                |           |           |           |    |    |
|-------------------|-----------|-----------|----|----|----|-------------------|-----------|-----------|-----------|----|----|
| $D \backslash DD$ | NB        | NS        | ZE | PS | PB | $D \backslash DD$ | NB        | NS        | ZE        | PS | PB |
| NB                | NB        | NB        | NB | NB | NB | NB                | NB        | NB        | NB        | NB | NB |
| NS                | NB        | NB        | NB | NB | NB | NS                | NB        | <b>PB</b> | <b>PB</b> | NB | NB |
| ZE                | <b>PB</b> | <b>PB</b> | ZE | NB | NB | ZE                | <b>NS</b> | <b>PB</b> | ZE        | NB | NB |
| PS                | NB        | NB        | NB | NB | NB | PS                | NB        | NB        | NB        | NB | NB |
| PB                | NB        | NB        | NB | NB | NB | PB                | NB        | NB        | NB        | NB | NB |

Table 3.3: Initial rule base for VL and VS in an adaptive controller

Only shortly before  $t = 160s$ , when the value of the rate is reaching machine precision and therefore is considered as 0, adaptation goes on as can be seen in Figure 3.4. A part of this problem can be solved easily, by applying a threshold value in stead of an exact 0:

- If  $D < 0$  and  $DD \leq 10^{-2}$  then raise the conclusion part of the most triggered rule with 1
- If  $D > 0$  then lower the conclusion part of the most triggered rule with 1

Of course the value this threshold should have is a point of consideration, however because this problem does not occur very often it is not that important.

With these slightly adjusted rules we do not solve the whole problem as shown in Figure 3.6. Here the first part of the problem still applies: the small valve is already fully opened, but the most triggered rule for *VL* is still the  $ZE \& ZE \rightarrow ZE$  rule so nothing happens. There are several ways to deal with this problem: changing the membership functions, dropping the  $ZE \& ZE \rightarrow ZE$  condition or adding extra rules to change the *VL* rule base when the *VS* rule base has come to a limit (i.e. NB or PB). The first option, changing the membership functions seems the most promising and the least dangerous.

### 3.3 Input scaling

As we saw in §3.2, Table 3.3 the rule base of the adaptive controller did not adapt over its full range due to bad scaling of the input  $DD$  (no scaling that is). The level can change with a rate roughly between  $-0.3m/s$  and  $0.9m/s$  due to the design of the water tank. This means a scaling of the  $DD$  input could be to multiply the (crisp) value of  $DD$  with a factor 3 if the rate is negative and leave it as is when it is positive.

Although the level can vary between 0 and 2 so  $D$  can vary between  $-2$  and  $2$ , the input  $D$  will not be (re)scaled because errors larger than 1 rarely occur and even if they do, they are dealt with just as well with the current scaling.

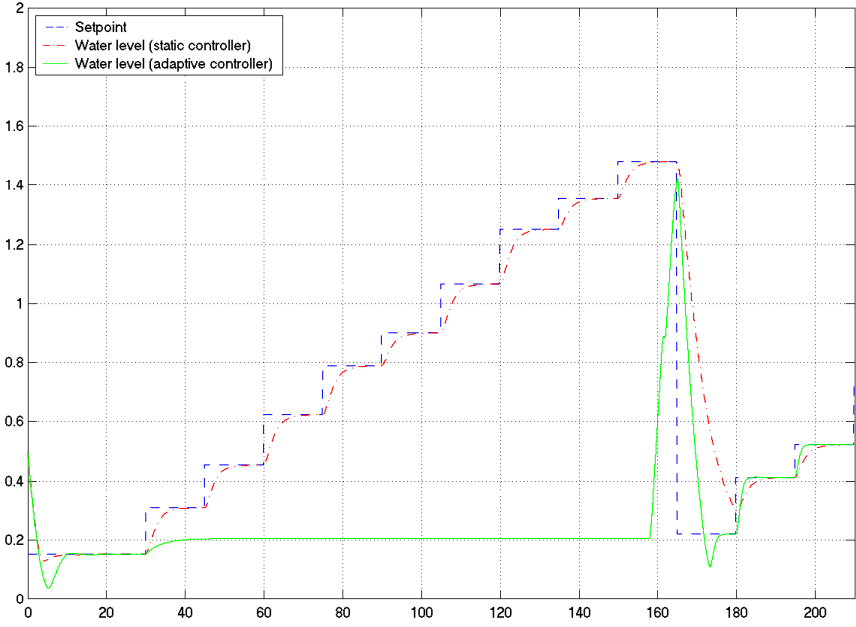


Figure 3.4: The adaptation algorithm gets stuck.

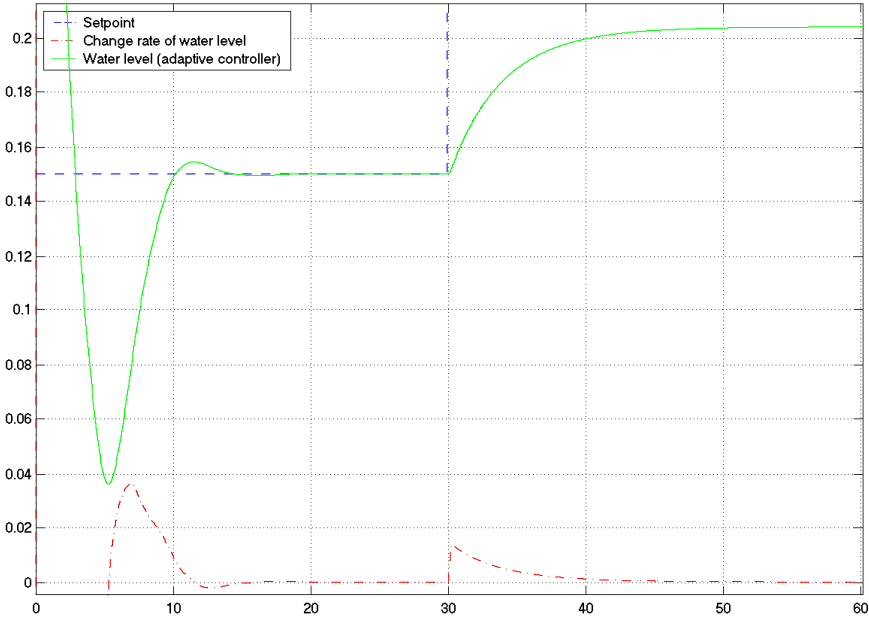


Figure 3.5: The change rate of the water level is decreasing but still positive.

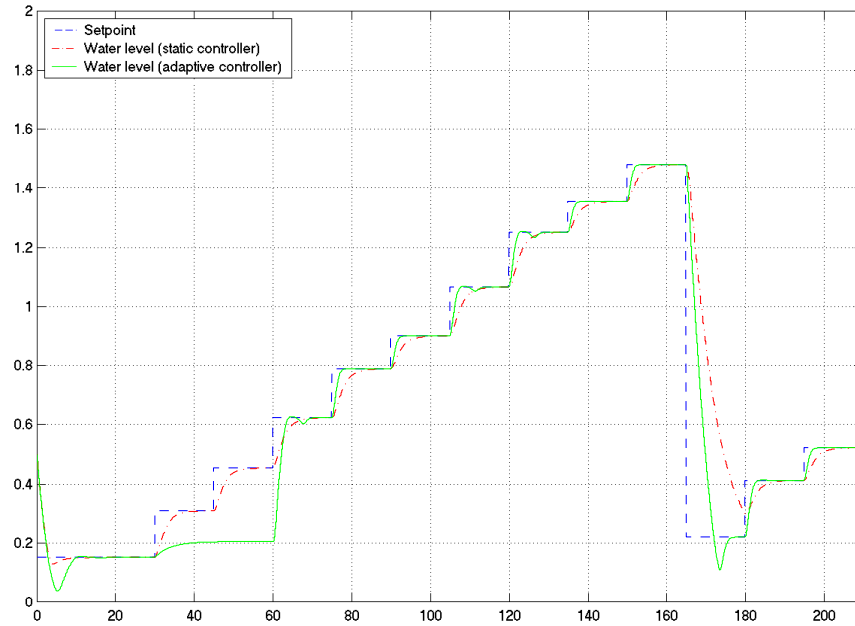


Figure 3.6: The adaptation problem is solved for the major part.

The performance of the adaptive controller using this extra scaling is much better, as can be seen in Figure 3.7. The only problem with this kind of scaling is that it makes the controller much more problem specific. A better way would be to scale inputs dynamically, or to have an enormous input range and then dynamically set the positions of membership functions within a meaningful area.

### 3.4 5 versus 7 Membership Functions

A comparative study has been carried out to investigate the difference in performance of the adaptive controller when using 5 or 7 membership functions. This was not done for the static controller because there is no ‘optimized’ rule base available for 7 membership functions. Also the gain the static controller would have is expected to be less, because the controller already has a very gradual rule base.

Both controllers seem to perform quite well (see Figure 3.8). The difference between the controller using 5 MFs for each input and the one using 7 MFs is very small. The one using 7 MFs seems to have a little more difficulties with initialization (there is a little dip at  $t = 30s$ ), is a little bit slower and has less overshoot and undershoot.

Evaluating the 7MF controller costs approximately 25% more time than evaluating the 5MF controller. This is a substantial increase in calculation time, because this evaluation costs roughly 20% of the total simulation time. With this small dif-

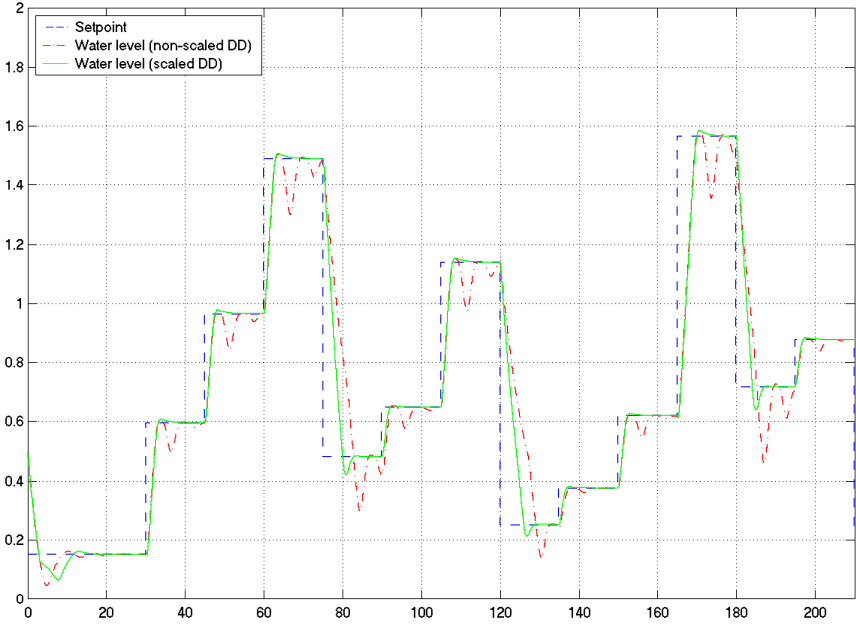


Figure 3.7: Scaling the water level change rate removes the fluctuations.

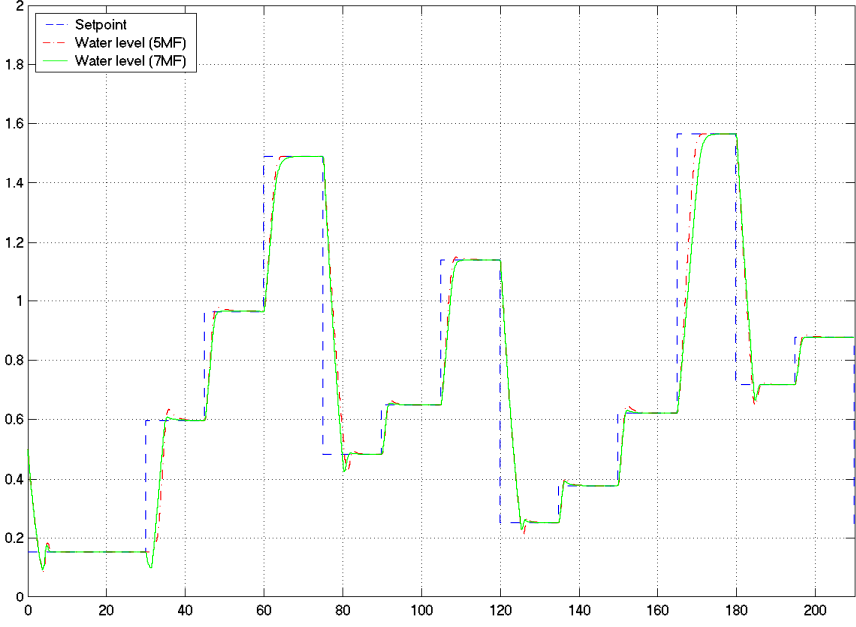


Figure 3.8: Controller performance using 5 or 7 membership functions per input.

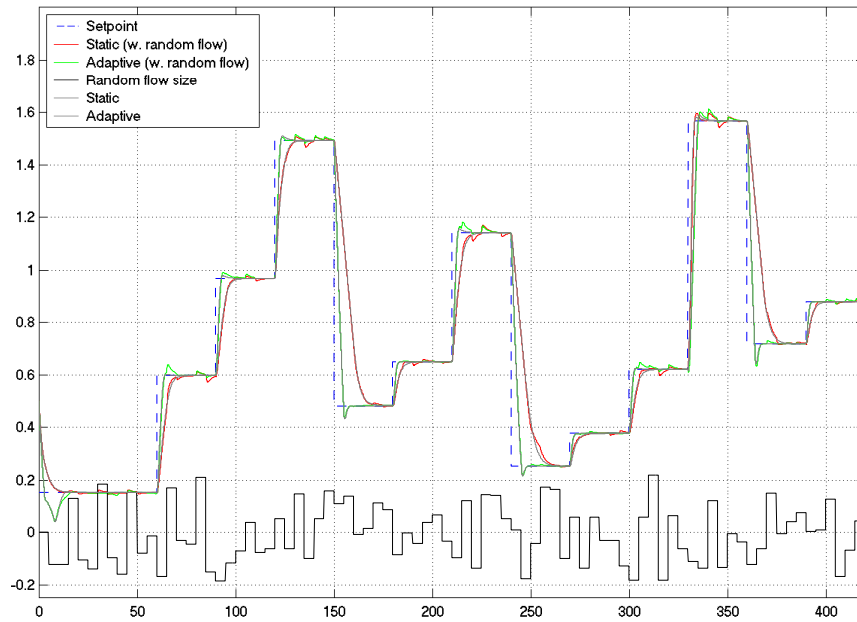


Figure 3.9: Controller performance with randomly changing flow rates.

ference in performance, the increased complexity and the higher evaluation time, the implementation of 7 MFs does not seem to pay off.

### 3.5 Performance with Changing Flow Rates

To test the adaptive controller for robustness, tests have been carried out with a changing maximum flow rate through the refilling pipes. In normal circumstances the maximum flow rate for the small flow is 0.1 and 0.8 for the large flow. Now we increase or decrease these flow rates with a values which change over time and can run up to 25% of the original maximal flows. In the first test this value is chosen randomly every few seconds, in the second test the change is sinusoidal to resemble the slow and continuous changes in reactor physics.

Figure 3.9 shows that both the static and the adaptive controller still do their work properly. The adaptive controller still has a much faster response and is still able to maintain a relatively stable water level at the setpoint value.

To measure the influence this distortion has on the controller, the difference between the water level in a system without changing flows has been compared to the level in a system with these random flows. For a long run (1000 seconds) the mean difference in water level and the mean percentage difference are calculated. The percentage difference is interesting because it compensates for the fact that for high water levels the valves will be opened wider, so a sudden change in flow will have a larger effect.

|          | Mean difference<br>Random flow changing every |        |        | Mean percentage difference<br>Random flow changing every |      |      |
|----------|---|--------|--------|--|------|------|
|          | 1s  | 2s     | 5s     | 1s   | 2s   | 5s   |
| Static   | 0.0110  | 0.0102 | 0.0069 | 1.47   | 1.33 | 1.21 |
| Adaptive | 0.0210  | 0.0102 | 0.0054 | 3.44   | 1.43 | 0.81 |

Table 3.4: Robustness of the static and adaptive controller to random flow changes.

|          | Mean difference<br>Sine frequency |           |           | Mean percentage difference<br>Sine frequency |           |           |
|----------|-----------------------------------|-----------|-----------|--|-----------|-----------|
|          | 0.1rad/s                          | 0.05rad/s | 0.01rad/s | 0.1rad/s                                     | 0.05rad/s | 0.01rad/s |
| Static   | 0.0068                            | 0.0048    | 0.0029    | 1.16   | 0.87      | 0.58      |
| Adaptive | 0.0060                            | 0.0028    | 0.0018    | 1.34   | 0.44      | 0.27      |

Table 3.5: Robustness of the static and adaptive controller to sinusoidal flow changes.

Table 3.4 shows that the adaptive controller needs some time to adapt itself to every flow rate change: when the flow rate changes every 5 seconds the adaptive controller is more robust than the static one, but when the changes come faster after each other the static controller is more robust. To the more gradual flow changes of the sinusoidal change the adaptive controller is more robust as Table 3.5 shows.

It is important not to mix up robustness with performance. Robustness only tells how close the controller performance will stick to the original performance when an external distortion is applied. Therefore it can well be that the one controller is less robust, but still has (far) better performance.

Both controllers appear quite robust to this distortion. For a better judgment on the controller robustness some other distortions which perhaps are more representing the actual distortions in nuclear reactors should be applied.



## Chapter 4

# Application of Neural Networks

Neural Networks are very useful because of their learning capabilities. When NNs are trained with some known data sets they are not only able to reproduce the data, but can also approximate outputs for other inputs than the trained ones. This makes NNs very suitable as a response surface or curve fitter. This way a NN could mimic the plant (the system to be controlled) thus enabling off-line tuning and tests or even model reference control or model predictive control. Another application is to use the NN for ‘training’ a neuro-fuzzy controller with data representing ‘good control’.

Applying neuro-fuzzy or neural network control and modeling should be possible according to literature ([29], [30], [31], [32]). However detailed information on implementation of these techniques is hard to find and the results of the current research were not fully satisfying. For further research in this direction it is advisable to try to find another (third party) toolbox or papers on implementations for a similar problems first.

### 4.1 Neural Network Plant Models

#### 4.1.1 Background

A problem with the setup as used in Chapter 3 is that in the nuclear reactor case safety regulations make it impossible to let a controller adapt itself on-line (i.e. when it is actually controlling the reactor)(Figure 4.1(a)). Therefore it is important to obtain a good model of the plant so the controller can be adapted off-line (Figure 4.1(b)) which is fully safe and the adapted controller can be checked for stability by engineers before it is actually implemented on-line.

It is difficult to obtain a good and fast model of a system, especially when it’s dynamics are not fully known. This is often the case, because even if a mathemat-

ical formulation exists, most of the parameters in that formulation are uncertain or even change during operation. A neural network could be trained to mimic the plant's behavior (Figure 4.1(c)) without any mathematical knowledge of the process and thus is a very general solution for modeling a plant's behavior.

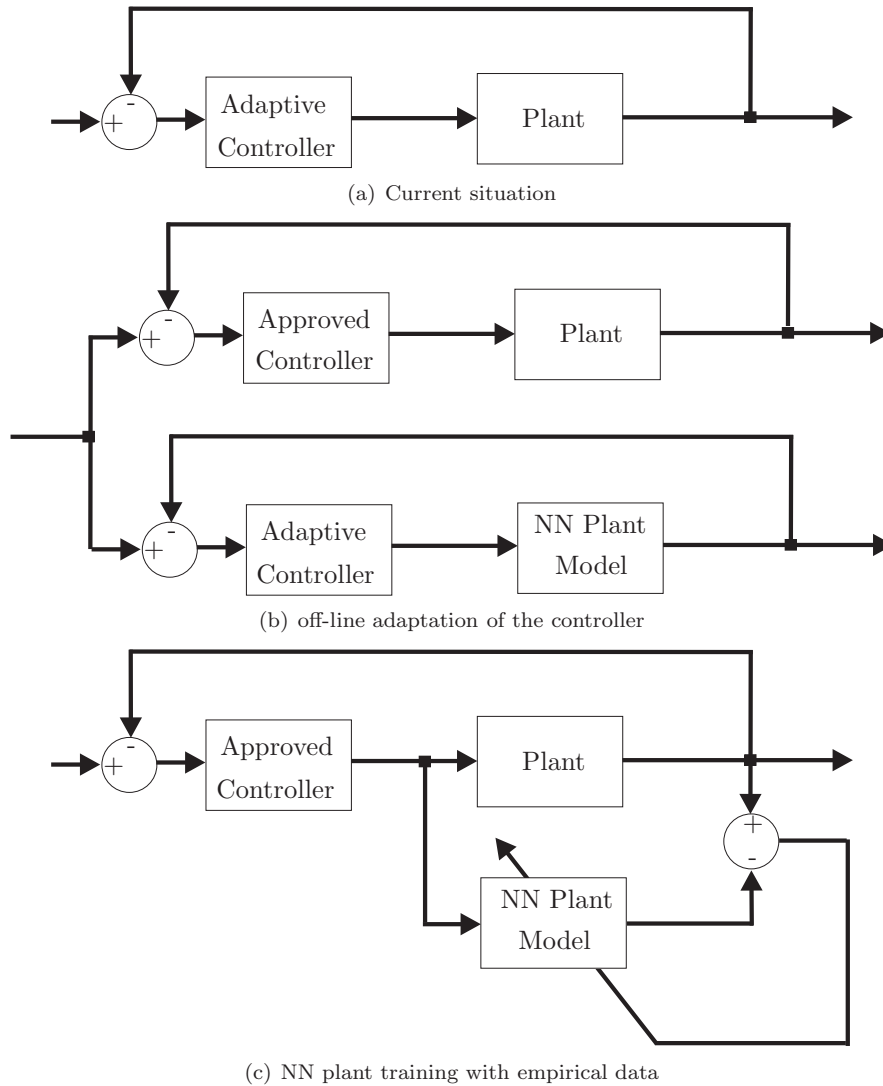


Figure 4.1: Global implementation setup NN based adaptive controllers.

The plant should approximate the next state given the current state and the control outputs. For NN training this means the input sets consist of the current water level ( $D_t$ ), the current water level change rate ( $DD_t$ ), the opening/closing speed of  $VL$  ( $VL_t$ ) and the opening/closing speed of  $VS$  ( $VS_t$ ) and the accompa-

nying output sets consist of the new water level ( $D_{t+dt}$ ) and the new water level change rate ( $DD_{t+dt}$ ). Of course it is possible to only estimate  $D_{t+dt}$  or  $DD_{t+dt}$  and derive the other one from the previous state. This might decrease training time and increase performance.

The performance of the networks is tested by feeding them with the signals of a static controller, basing its control strategy on the state of an official tank simulation. For real tests of course the controller should base its decisions on the estimated state, but then a good result would be far too difficult to get using the currently trained networks.

### 4.1.2 Neural Networks

A lot of different types of NNs are available in MATLAB. Some are for data classification only and therefore not suited to mimic the plant, but most types should be well capable of doing so. Three groups of NNs have been investigated:

1. Back Propagation networks
  - Feed-Forward back propagation (FF)
  - Cascade-Forward back propagation (CF)
  - Elman back propagation (ELM)
2. Radial Basis networks
  - standard Radial Basis (RB)
  - Generalized Regression (GR)
3. Neuro-Fuzzy networks
  - Adaptive Neuro-Fuzzy Inference Systems (ANFIS)

These networks have been trained with different settings (such as number of neurons, number of hidden layers and spread factors for RB and GR NNs) and for different data set sizes (2500, 5000 and 10000 data pairs). Also a estimation of only the level ( $D_{t+dt}$ ) has been tried.

#### Back Propagation networks

When we check the performance of the networks with a test-data set, the CF networks come out best, the FF networks come second and the ELM networks worst. For the CF and FF networks, the heavier networks (more layers, more neurons, more data-pairs) are the best (like one would expect), but lighter networks with small sets still outperform heavy networks with small data sets and light networks with large datasets. This would mean that large networks are better off with large sets and small networks are better off with small sets. For ELM networks things only get worse when more neurons are added.

When we check the performance using the estimation error in an actual control simulation, things turn out very different. The ELM networks still perform

miserable, but now CF and FF networks are much more mixed and small data sets seem to be the way to go, preferably in combination with a light network. However, the performance of the best network (CF, 10 neurons, 2500 data pairs, performance see Figure 4.2) is still too poor to be really useful.

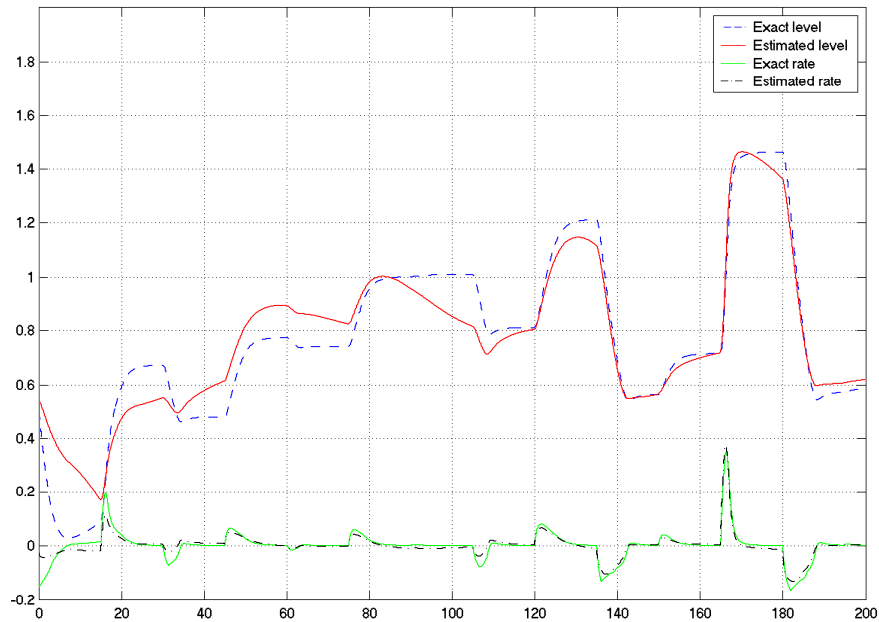


Figure 4.2: Simulation using the best tested network providing a level and rate estimation.

When only estimating the level things are totally different again, although the CF networks are the over-all winners again. In the test now middle-class networks with small training sets perform best. Based on the simulation this is almost the same case, though a two-layer 2x5 neuron network with a large (10000 data pairs) training set has the smallest error. Performance plots of the two best networks according to the simulation are provided in Figures 4.4 and 4.3.

The training needed now is about 2/3 of the time needed for training both level and rate.

Although the shapes of the response curves are reasonable, the exact values are by far not accurate enough to be applied in a real system. This may also be due to the fact that in the simulation errors accumulate and the fact that neural networks generally provide an approximation instead of an exact answer. It might thus even be impossible by principle to make an accurate plant simulation model.

No straight conclusions about the network layout or number of training sets can be formulated from these tests. The only clear thing is that CF networks are the best choice when considering BP networks.

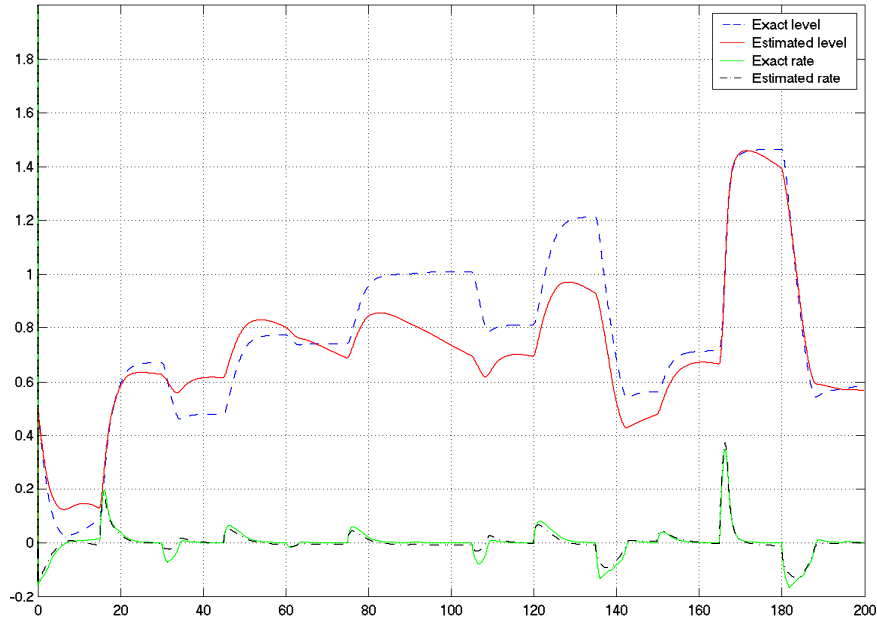


Figure 4.3: Simulation using the best tested network providing a level only estimation (CF, 2 layer, 2x5 neurons, 10000 data pairs, rate is derived from level information).

### Radial Basis networks

It is very difficult to get normal RB network to mimic the plant accurately. When using too many data sets the performance gets worse, probably due to conflicting data (almost the same inputs, but different outputs). It is therefore important to make a well-balanced dataset and choose the appropriate spread factor (first hidden layer bias value). A result can be seen in Figure 4.5, where 2500 datasets and a spread value of 0.9 have been used to make the network.

When the adaptive system is used (adding neurons one by one) results do not get any better and training takes a lot longer. Also training the level only comes with some problems: when the flow has stabilized, the NN does not see that the rate is 0 and estimates a slightly positive or negative value, thus changing the level when it should be steady.

The GR network seems to give better results (Figure 4.6). The shape is right and the lines are straight, the only problem is that the level is always a certain amount too high. This is caused by the bad estimation in the beginning, probably because the training data contains to few sets with these very low levels. But if we would eliminate this first error we get the plot in Figure 4.7, and we see that the response would still not be accurate enough to design a controller upon.

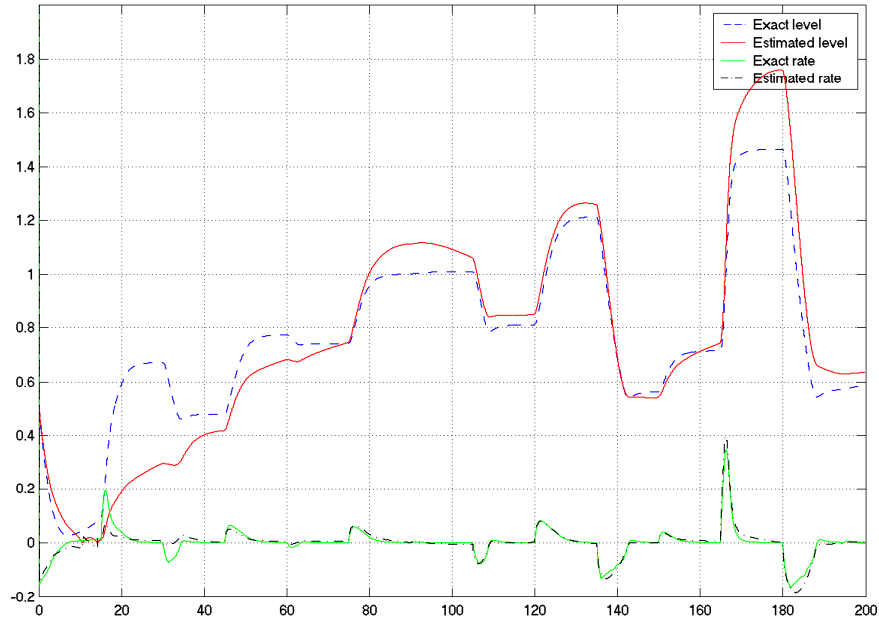


Figure 4.4: Simulation using the second best tested network providing a level only estimation (CF, 2 layer, 20 resp. 10 neurons, 2500 data pairs).

### Adaptive Neuro-Fuzzy Inference Systems

About the ANFIS we can be short: it does not work at all. None of the ANFISes trained made any sense in a control simulation. This means that the type of ANFIS supported by MATLAB is inadequate, the current implementation is incorrect or the whole idea of using an ANFIS for this purpose is wrong.

## 4.2 Neuro (Fuzzy) Controllers

The Fuzzy Logic Toolbox of MATLAB provides a tool to support the design of an ANFIS (Adaptive Neuro-Fuzzy Inference System) controller. An ANFIS is a true hybrid system where no clear distinction can be made anymore between the fuzzy logic controller and the neural network. The controller can be fed with training data that reflect 'good control'. This is immediately the problem: if we already know 'good control' for our system, we don't need such a system anymore.

ANFIS could be used very well to convert the expert knowledge of a manual operator into an automated controller. Another option would be to try inverse model control. This theory is based on the idea that if we know the transfer function of our system, we can control it in an exact manner if we make a controller with the exact inverse transfer function. In classical control this is impossible

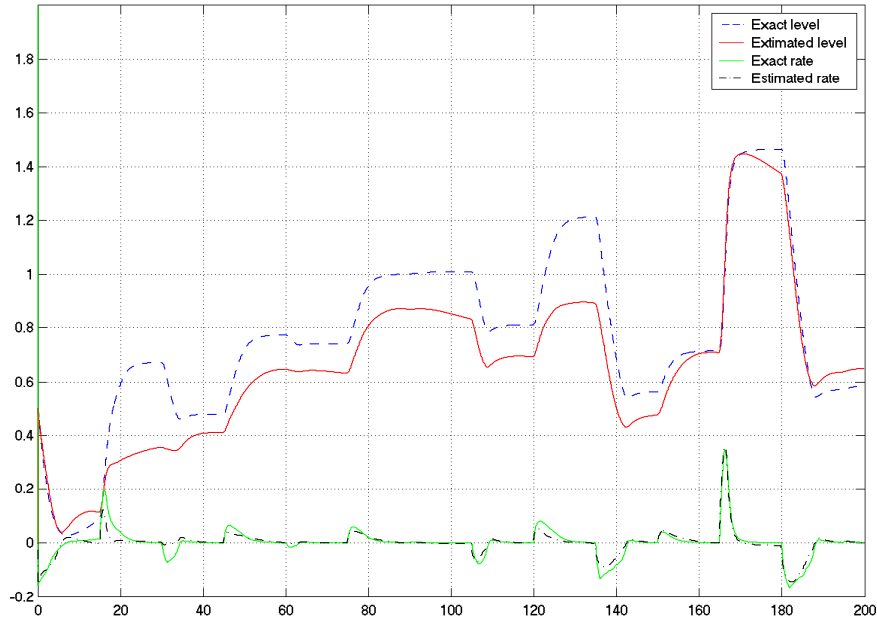


Figure 4.5: Simulation using the best tested RB network providing a level and rate estimation.

because these inverse transfer functions do not exist in practice, but it should be possible by training an ANFIS with the systems output (and previous state) as input and the systems input as ANFIS output. This is however quite like training an ANFIS to mimic the plant, only should it mimic the inverse plant, it is therefore quite logic that this did not work. A problem is that there are no examples of implementations of similar control systems, which makes it difficult to find out whether this principle should be applied in a different way or that it is just a bad or wrong implementation.

Other neural controllers can be found in the Neural Network Toolbox. These controllers are pure neural networks, so there is no fuzzyness involved. This will make it even more difficult to implement such a controller in a real nuclear reactor, because neural networks are much less transparent than fuzzy systems and thus more difficult to check for safety.

A problem with these controllers is that one of them even does not work in the official MATLAB demo and that they all give trouble when the training process is near completion, due to an unclear bug in the original MATLAB implementation. Also for proper training of a complex system like the water tank demo, a lot of datasets are needed, which makes the training terribly slow and even may cause the computer to crash.

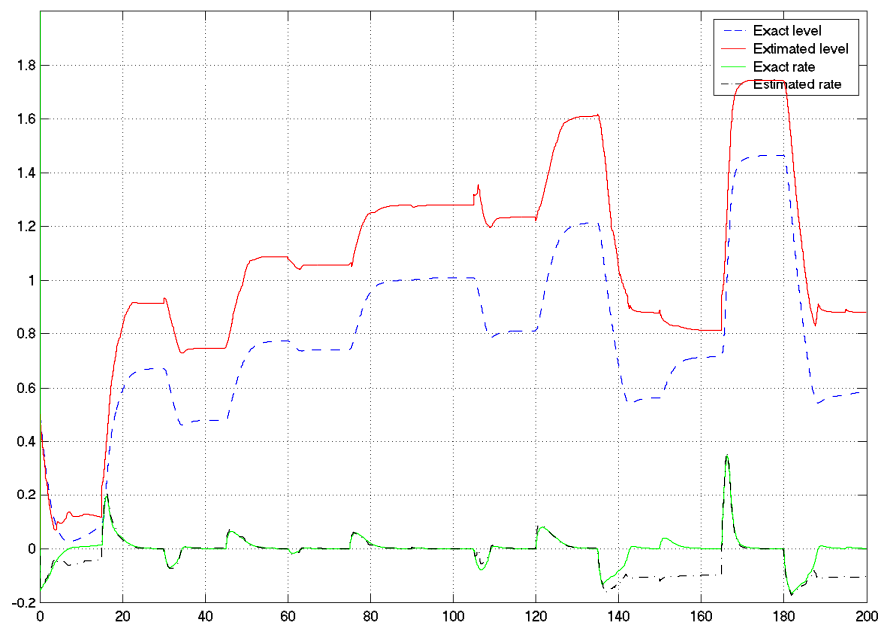


Figure 4.6: Simulation using the best tested GR network providing a level and rate estimation.

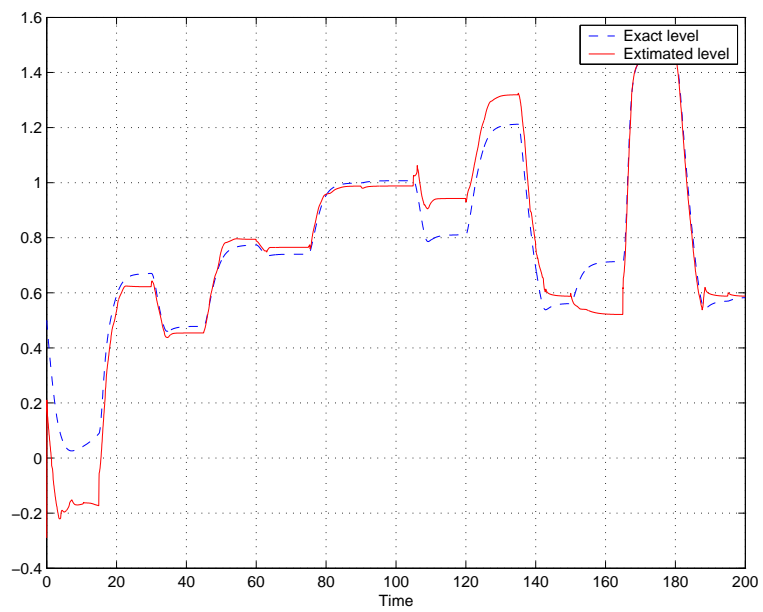


Figure 4.7: Edited version of Figure 4.6 to compensate for the error originating from the period between 15-20s.



## Chapter 5

# Genetic Fuzzy Optimization

Genetic algorithms are very general optimizers. They can handle very complex and coupled systems without the need of derivative information to define a search direction. Also genetic algorithms can very well handle search spaces with a lot of local optima. Therefore they can very well be applied to optimize the performance of a fuzzy logic controller for an arbitrary system.

From MATLAB [6] release 14 on a genetic algorithm toolbox is included standard. This toolbox was not yet available at the time of this research, so a third party GA toolbox called GAOT [22] was used. This GAOT toolbox actually has more extensive capabilities, but it does not include a graphical user interface.

### 5.1 Genetic Fuzzy Optimization

There are a lot of parameters in a FLC that can be tuned to improve its performance. To mention a few:

- the rules
- the positioning of the membership functions in the input and output space
- the weights applied to each rule
- the shape of the input and output membership functions
- the number of membership functions

The rules are already tuned by the adaption algorithm, which appears to work fairly well. The positioning of the membership functions within the input space seems an interesting topic because it could make the (adaptive) controller be much more problem independent. Adapting the weights of the rules could be interesting, but a lot of parameters would be needed (which very much slows down the GA). The other possible tuning parameters often do not make much of a difference, which makes the positioning of the MFs the best choice.

### 5.1.1 Genetic Parametrization

To optimize the membership functions, a set of parameters must be found that satisfyingly defines their positions in the input space. Satisfyingly here means that we want as few parameters as possible and the parameters must still have as much ‘physical relevance’ as possible, to make the crossover operator a meaningful tool to reach convergence. Also the parametrization must prevent ambiguity (switching the positions of MFs) and practically impossible or unwanted variants.

The parametrization chosen uses the ‘remain range’, the space left over between the MFs that are already set and the input ranges, to determine the positioning of the tops of the MFs (see Figure 5.1). A few constraints were added to the parametrization:

- The central MF always has its top at  $x = 0$
- The  $x$  position of the top of one MF is one of the base point for the neighboring MFs (so no gaps between MFs can arise)
- The MFs for  $VL$  and  $VS$  are the same, they are just scaled down for  $VS$  (like in the static controller case).

Some optimizations with independent MFs for  $VL$  and  $VS$  have been performed, the results however were not better and often even worse than optimizations with the same MFs for  $VL$  and  $VS$ . This could be due to the fact that the parameter vector is twice as long in this case, which makes it more difficult to for the GA to converge.

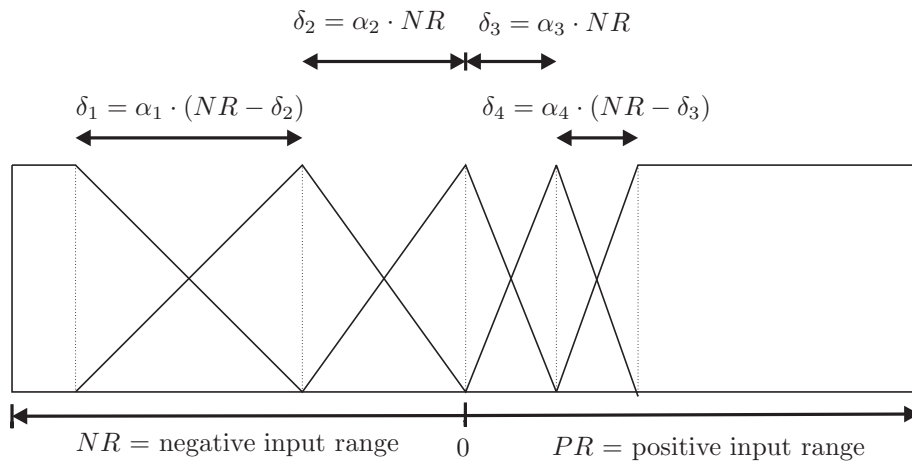


Figure 5.1: Derivation of the membership functions from a parameter vector  $\alpha$ .

The GA will create different individuals (possible solutions) using this parametrization and apply crossover, mutation and selection to reach an optimum. Actually the individual is only a vector of 8 values: 4 values for each of the 2 5MF inputs.

### 5.1.2 Fitness Function

The fitness function is the heart of the genetic algorithm, because it defines how well each individual performs. The optimum obtained by the GA is highly dependent on the fitness function, therefore it has to be designed with care. It is important to make an inventory of all characteristics of good (or bad) controller performance, but also a representative control simulation has to be designed.

Designing a representative control simulation is not trivial: the problems the controller will come across during actual operation should all be present in the simulation in the right proportions. At the same time the simulation should be as short as possible to save time. For actual application some more time should be invested in the simulation design (i.e. the setpoint changes over time), also some checks on robustness might be added.

To see whether a certain simulation or fitness function is general enough to cover the whole operation range of the system, it might be a good idea to design multiple simulations and compare the resulting fitnesses for some different controllers. A better fitness according to the one simulation should also have a better fitness according to the other, if both simulations are considered ‘meaningful’. When plotting the fitnesses calculated with both simulations against each other, a straight line would be the best. This check on the meaningfulness of the simulation has been performed on some controllers saved by the GA during an optimization (see Figure 5.2). The simulations both use a uniform random value as setpoint, only the random seed is different for the original GA simulation and the test simulation. For ‘just a random value’ the meaningfulness seems still quite good.

At the moment some different fitness functions have been implemented. These differences can be found in the simulation time (30 versus 250 seconds) and in the fitness criteria:

- **The simple fitness functions** only use the error area (difference between water level and setpoint integrated over time) as fitness value, where positive errors (i.e. overshoot) are multiplied by a factor larger than 1 because they are considered to be worse because of their possible dangerous consequences in a nuclear reactor.
- **The extensive fitness functions** also consider the mean settling time, mean overshoot, mean positive overshoot and mean negative overshoot, each with a different weight.

### 5.1.3 Results

The drawback of a GA is that it generally needs a lot of generations (iterations) to converge. In this case, with the time consuming SIMULINK simulations, this means that an optimization may take over an hour. Several optimization runs are needed to draw conclusions on the GA’s behavior because the randomness enclosed in the principle of the GA gives every optimization run a different twist. This makes it

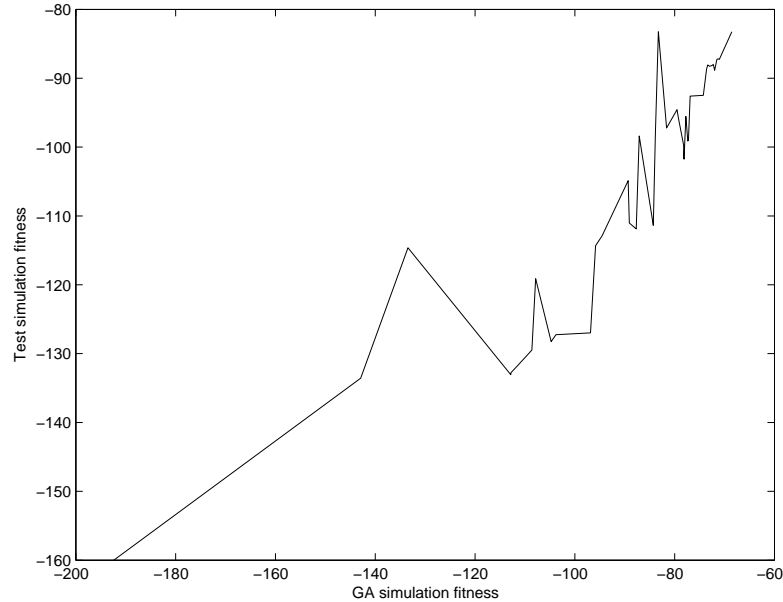


Figure 5.2: Fitness values obtained for some FLCs using two different simulations with a differently randomized setpoint.

difficult to check the overall convergence of the GA and tweak its parameters for better, more robust performance.

Despite the fact that tweaking the GA may be difficult, some nice results can be obtained (even when still sub-optimal). The improvement reached can be seen in Figure 5.3 where the original (slow) response of the static controller and the response of the controller with optimized MFs are both plotted. In Figure 5.4 the optimized membership functions can be seen. The original MFs are depicted in Figure 3.1. For this optimization the extensive fitness function was used.

In this response we see the speed of the controller has improved significantly, at the cost of a little overshoot. The overshoot could be suppressed more by applying a higher weight to the overshoot in the fitness function. This is of course decision that should be made according to the wishes of the controller designer and the specifications given.

When we look at the MFs for the level (the upper ones in Figure 5.4) we see that the center MFs are put closer together. This is not very surprising, because most control actions will be in this range and surely the accuracy in this range will be more important. When looking at the MFs for the rate (the lower ones) the most eye-catching thing is the disappearance of the *NB* membership function. This can very well be due to the fact that there is not much of a change in the rule between  $DD = NS$  and  $DD = NB$  (see §3.1). Also the fact that rates smaller than  $-0.3$  are physically impossible might play a role here.

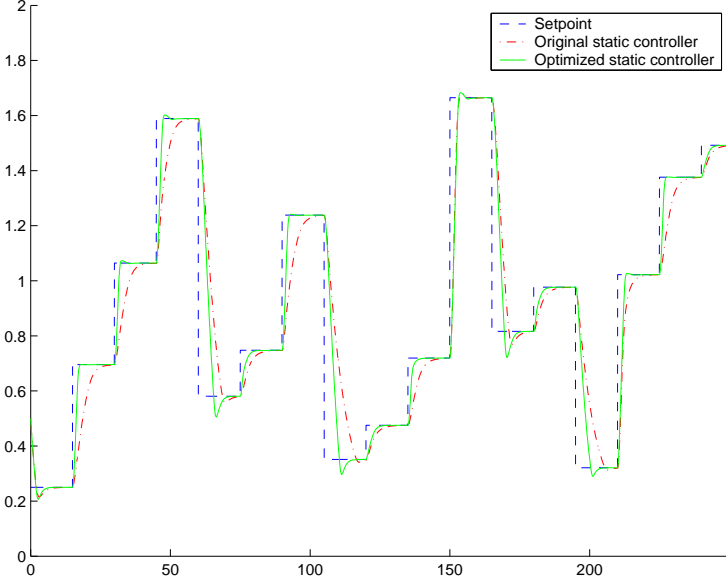


Figure 5.3: Response of the original and the optimized controller.

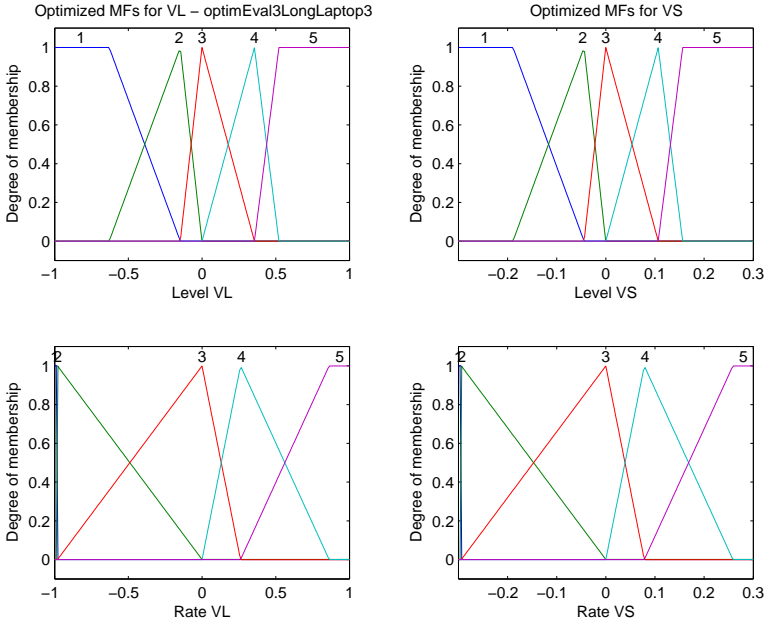


Figure 5.4: Optimized membership functions for the inputs of VL and VS.

## 5.2 Genetic Adaptive Fuzzy Optimization

For the optimization of the adaptive controller the parametrization and the extensive evaluation function as described in §5.1 is used. The main difference is that the optimization has now become an iterative process, where the controller first gets some time to adapt (using the adaptation mechanism from §3.2), after which the membership functions are optimized, followed by a new adaptation run and a new optimization run etc. This process is likely to converge very fast, because the rule base and the MFs keep the same mutual relation.

Because the optimization will be carried out 2 or 3 times now, lower demands can be made of the convergence of the GA. If the GA has not fully converged in the first run, it can carry on in the second run. To make this principle work, the best found solution from the last optimization should be added to the initial population of the new GA run.

Figures 5.5 and 5.6 show that the genetic optimization is able to remove the fluctuations in the original adaptive controller without doing much harm to the rest of the performance (there is even no overshoot anymore). This means the optimization algorithm provides a very general way to improve controller performance, because engineering knowledge like applied in §3.3 is not needed any longer. The improved MFs (Figure 5.7) show the same pattern as in the static case (Figure 5.4).

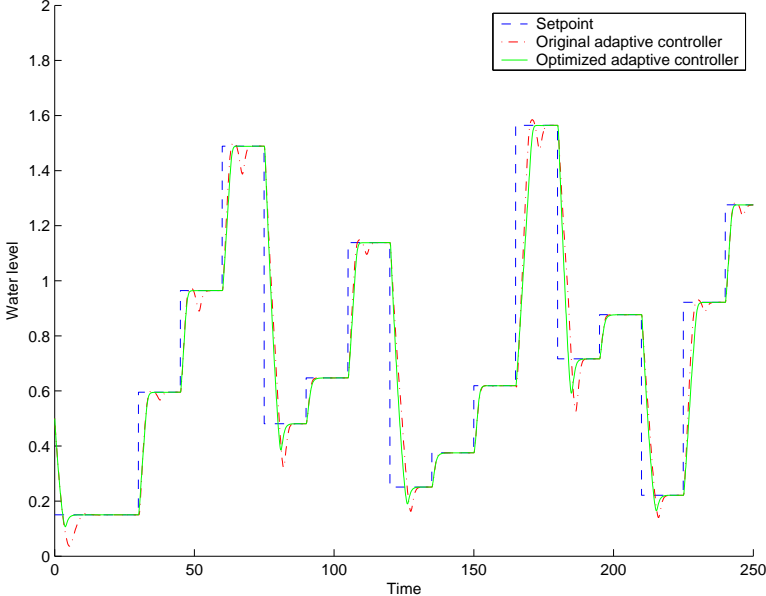


Figure 5.5: Response of the original and the optimized controller.

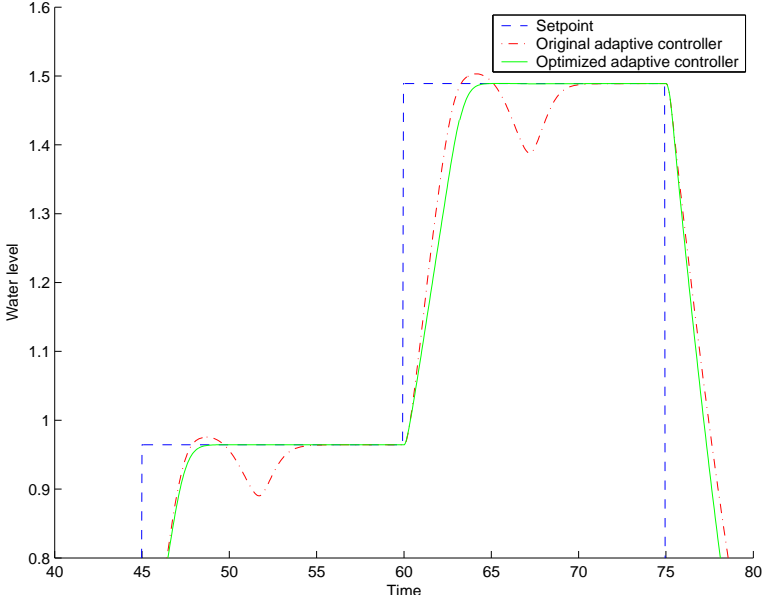


Figure 5.6: Cut out of the original vs. the optimized controller response.

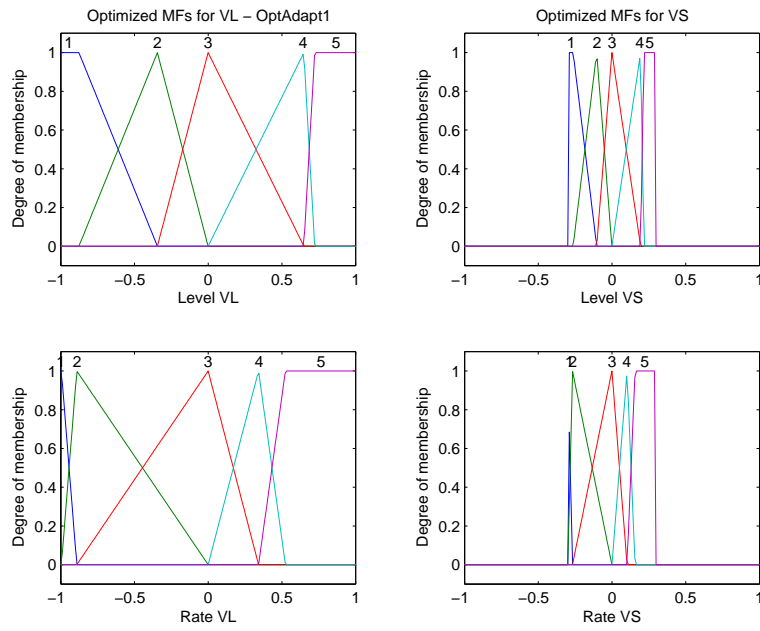


Figure 5.7: Optimized membership functions for the inputs of  $VL$  and  $VS$ .

## Chapter 6

# Conclusion

When dealing with soft computing techniques it is difficult to make hard statements on performance, due to a lack of in-depth investigations at the current stage. Actually the vagueness which is the strength of these techniques is also their weakness when it comes to drawing conclusions. However some very meaningful conclusions can be drawn from the research done.

As shown by almost any test in this report fuzzy logic provides a powerful way to control a strongly non-linear system, even when the system is not fully defined or has changing properties. Although the strength of the applied soft computing methods would be that no detailed (mathematical) description of the plant would be needed, even when such a description is available FLC still has many benefits, such as its robustness and transparency.

A static controller (§3.1) is the simplest and safest solution in an industrial environment, however designing the rule base for a static Fuzzy Logic Controller (FLC) by hand costs a lot of time and engineering skills. Moreover, such controllers are very problem specific and are likely not to provide optimal control in real life applications.

Adaptive control can solve these problems. For adaptive control very limited knowledge of the system is needed. Actually for the problem investigated only two simple ‘common sense’ rules are needed to fully control the strongly non-linear system, as shown in §3.2.

Tuning of the FLC is important and can be done by hand when some insight in the controller and the system to control is available. An automated tuning method using Genetic Algorithm optimization techniques was proposed and successfully tested in for the static controller case in §5.1 and in §5.2 for the adaptive controller. In both cases it appears to be very profitable to let the algorithm tune the controllers according to predefined desires such as minimum overshoot and short settling time.

For reasons of safety, any controller applied in a nuclear reactor should be extensively tested before being implemented. This actually means that every controller

should be a static controller. However, the way to get to that controller can be by adaptation and optimization, which provide a fast, easy and very general way to produce a dedicated high performance controller.

A major problem is that for adaptation and optimization an accurate model of the plant (the controlled system) is needed, because this cannot be done on-line. To maintain the problem independence and superfluity of a mathematical description of the plant, Neural Networks were proposed in §4.1 to obtain a simulation model for the plant. These however seem too inaccurate to be applicable. Perhaps more balanced training data, other or adjusted types of networks or better tuned networks can solve this problem, but finding a confirmation from literature that the proposed setup can work would be a first step in further research.

After all a promising solution has been found to make high performance FLCs for application in for instance nuclear reactor control. A subject for further research would be how to make the proposed method suitable for practical application in an industrial environment.

# Bibliography

- [1] J.A. Bernard. Use of a rule-based system for process control. *IEEE Control Systems Magazine*, 8(5):3–13, 1988.
- [2] D. Ruan. Initial experiments on fuzzy control for nuclear reactor operations at the belgian reactor 1. *Nuclear Technology*, 143:227–240, August 2003.
- [3] A.J. van der Wal D. Ruan. Controlling the power output of a nuclear reactor with fuzzy logic. *Information Sciences*, 110:151–177, 1998.
- [4] D. Ruan. Implementation of adaptive fuzzy control for a real-time control demo-model. *Real-Time Systems*, 21:219–239, 2001.
- [5] D. Ruan, editor. *Fuzzy Systems and Soft Computing in Nuclear Engineering*. Studies in Fuzzyness and Soft Computing. Physica-Verlag, 2000. ISBN 3-7908-1251-X.
- [6] Inc The MathWorks. Matlab 6.5, the language of technical computing. Software with online help (www.mathworks.com), 1984-2002.
- [7] D. Ruan, editor. *Intelligent Hybrid Systems: Fuzzy Logic, Neural Networks, and Genetic Algorithms*. Kluwer Academic Publishers, 1997. ISBN 0-7923-9999-4.
- [8] L.A. Zadeh J. Yen, R. Langari, editor. *Industrial Applications of Fuzzy Logic and Intelligent Systems*. IEEE Press, 1995. ISBN 0-7803-1048-9.
- [9] M. Sugeno H.T. Nguyen, editor. *Modeling and Control*. Fuzzy Systems. Kluwer Academic Publishers, 1998. ISBN 0-7923-8064-9.
- [10] L. A. Zadeh. Fuzzy sets. *Information Control*, 8:338–353, 1965.
- [11] E. H. Mamdani. Application of fuzzy algorithms for simple dynamic plant. *Proceedings IEEE*, 121(12):1585–1588, 1974.
- [12] C. von Altrock. *Fuzzy Logic & NeuroFuzzy Applications Explained - The Practical, Hands-On Guide to Building Fuzzy Logic Applications*. Prentice Hall PTR, 1995. ISBN 0-13-368465-2.
- [13] W. Pitts W.S. McCulloch. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133, 1943.

- [14] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.
- [15] J. Hopfield. Neural networks and physical systems with emergent collective computational properties. *Proceedings of the National Academy of Sciences of the USA*, 79:2554–2588, 1982.
- [16] L. Ziemianski Z. Waszczyszyn. Neural networks in mechanics of structures and materials - new results and prospects of applications. *Proceedings of European Conference on Computational Mechanics, München*, 1999.
- [17] W.M. Jenkins. Neural network-based approximations for structural analysis. *Developments in Neural Networks and Evolutionary Computing for Civil and Structural Engineering*, pages 25–35, 1995.
- [18] W. Ruijter J.O. Entzinger, R. Spallino. Multilevel distributed structure optimization. *Proceedings of the 24th International Congress of the Aeronautical Sciences (ICAS), Yokohama*, 2004.
- [19] J.H. Holland. *Adaptation in natural and artificial systems*. Ann Arbor: The University of Michigan Press, 1975. ISBN 0262581116.
- [20] S.E. Haupt R.L. Haupt. *Practical Genetic Algorithms*. Wiley-Interscience, 1997. ISBN 0-471-18873-5.
- [21] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989. ISBN 0-471-18873-5.
- [22] M.G. Kay C.R. Houck, J.A. Joines. *A Genetic Algorithm for Function Optimization: A Matlab Implementation*. GAOTv5 Manual (Genetic Algorithm Optimization Toolbox), 1998.
- [23] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer Verlag, 1997. ISBN 3-540-60676-9.
- [24] A. Afzalian D.A. Linkens. Training of neurofuzzy power system stabilisers using genetic algorithms. *International Journal of Electrical Power & Energy Systems*, 22:93–102, 2000.
- [25] S. Rizzo R. Spallino. Multiobjective discrete optimization of laminated structures. *Mechancis Research Communications*, Vol. 29, 17-25, 2002.
- [26] G. Giambanco R. Spallino, S. Rizzo. Shakedown optimal design of rc structures by evolution strategies. *Engineering Computations*, Vol. 17 No. 4, 440-458, 2000.
- [27] A. Marczyk. Genetic algorithms and evolutionary computation. <http://www.talkorigins.org/faqs/genalg/genalg.html>, 2004.
- [28] D. Beasley J. Heitkttter. The hitch-hiker’s guide to evolutionary computation. <http://www.cs.bham.ac.uk/Mirrors/ftp.de.uu.net/EC/clife/www/>, 2000.

- 
- [29] S.F. de Azevedo A. Andršik, A. Mszros. On-line tuning of a neural pid controller based on plant hybrid modeling. *Computers and Chemical Engineering*, 28(2004):1499–1509, 2003.
- [30] S. Mukhopadhyay J.F. Briceno, H. El-Mounayri. Selecting an artificial neural network for efficient modeling and accurate simulation of the milling process. *International Journal of Machine Tools & Manufacture*, 42(2002):663–674, 2002.
- [31] R. Babuška H.B. Verbruggen J.A. Roubos, S. Mollov. Fuzzy model-based predictive control using takagi-sugeno models. *International Journal of Approximate Reasoning*, 22(1999):3–30, 1999.
- [32] A. Dourado H. Duarte-Ramos P. Gil, J. Henriques. Fuzzy model-based predictive control using takagi-sugeno models. *Proceedings of ESIT'99 European Symposium on Intelligent Techniques, Crete, Greece, (1999)*, june 1999.



# Appendix A

## Implementation

An example SIMULINK model is shown in Figure A.1. Of course different implementations are needed for the different tests, but in general all schemes look like this one. The state of the system can be monitored using the scopes or the data saved to the MATLAB workspace.

Different patterns can be chosen for the setpoint value. In most tests the lower system is used, which generates a new random value (normally between 0.1 and 1) every 15 seconds, which is added to the current setpoint value. If the calculated setpoint surpasses 1.5 (+0.15) the setpoint will be decreased by 1.5. A ground-level of 0.15 is always maintained, which means that the setpoint can reach a maximum value of 1.65.

The subsystems ‘Static’ and ‘Adaptive’ are described in Appendix A.1 and the subsystem ‘Valves+Tank’ in A.2.

### A.1 Fuzzy Inference System (FIS)

As can be seen in Figures A.3, the FIS structures are evaluated and adapted by means of a m-file (§B.1.6 and §B.1.6). In some static cases a standard SIMULINK FIS block may be used because it is faster, but for the adaptive case this block does not have enough flexibility. This standard block may produce an error, a solution for this error is given in §A.3.

The FIS structures themselves can be made using the fuzzy editor, but for regular use it is more practical to make a script to do this (§B.1.1 and §B.1.2). The only difference between a static and a adaptive FIS is it’s rule base, which is therefore added to the FIS in a later stadium using a separated script (§B.1.4 and §B.1.3).

### A.2 Water Tank

Figure A.4 shows the basic idea of the water tank system. As a basis for the implementation, some SIMULINK blocks from the ‘water level control’ demo of

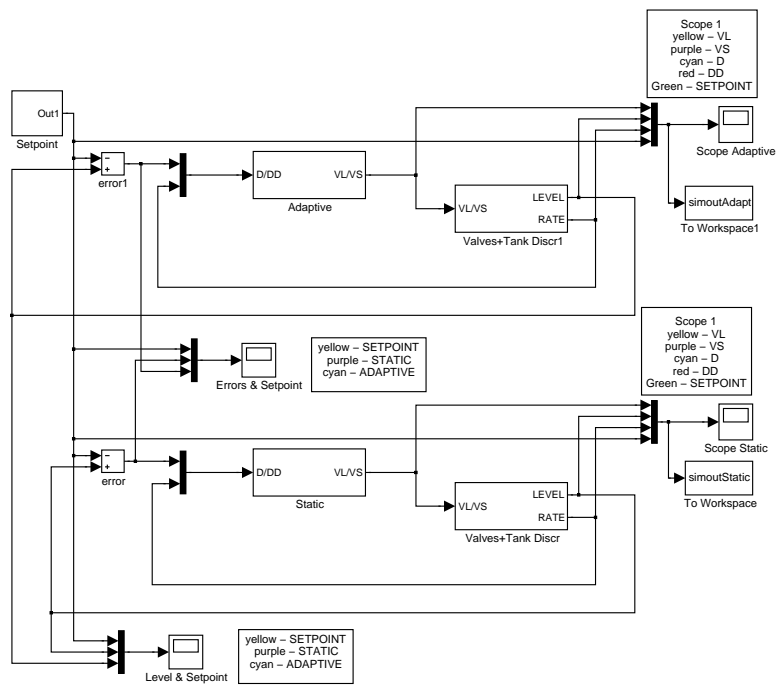


Figure A.1: SIMULINK implementation of the full system.

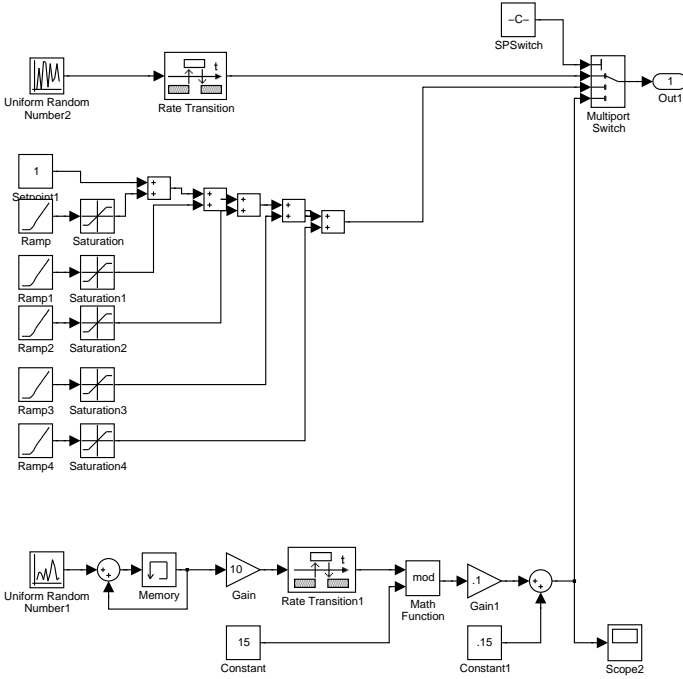


Figure A.2: SIMULINK scheme for generation of the setpoint value.

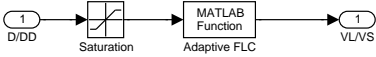


Figure A.3: SIMULINK scheme for the 'Adaptive' subsystem (static looks the same, only calls another m-file).

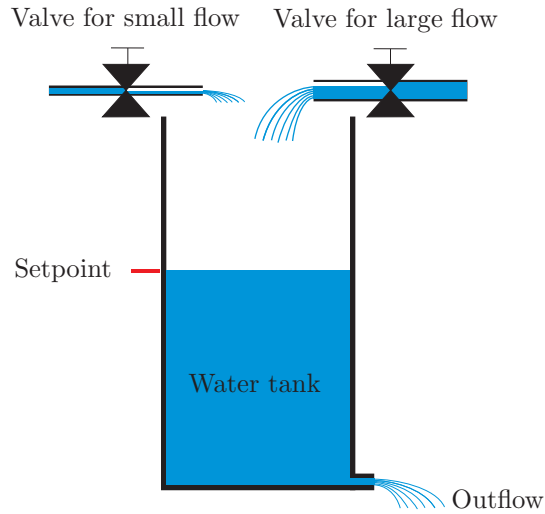


Figure A.4: Water tank demo model.

the MATLAB fuzzy logic toolbox were used. This resulted in the SIMULINK block diagram shown in Figure A.5. The underlying block diagrams of the valve block and the water tank block are shown in Figure A.6(a) and A.6(b) respectively.

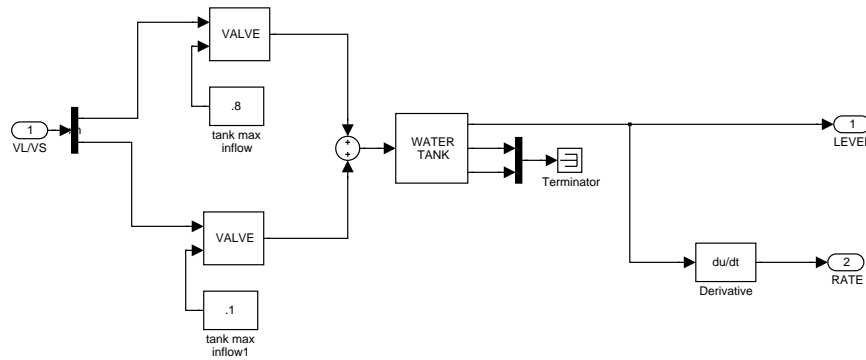


Figure A.5: SIMULINK implementation of the water tank demo model.

The integrator in the valves is limited to values between 0 (fully closed) and 1 (fully opened), both are initialized as fully closed. The height of the water tank is  $2m$ , the bottom area is  $1m^2$ , the cross-section of the outlet is  $0.05m^2$  and the initial water level in the tank is set to  $0.5m$ . The ‘Tank Volume’ block is an integrator limited between 0 and  $height \cdot area$ .

For faster evaluation and because the adaptive controller needs a constant sample

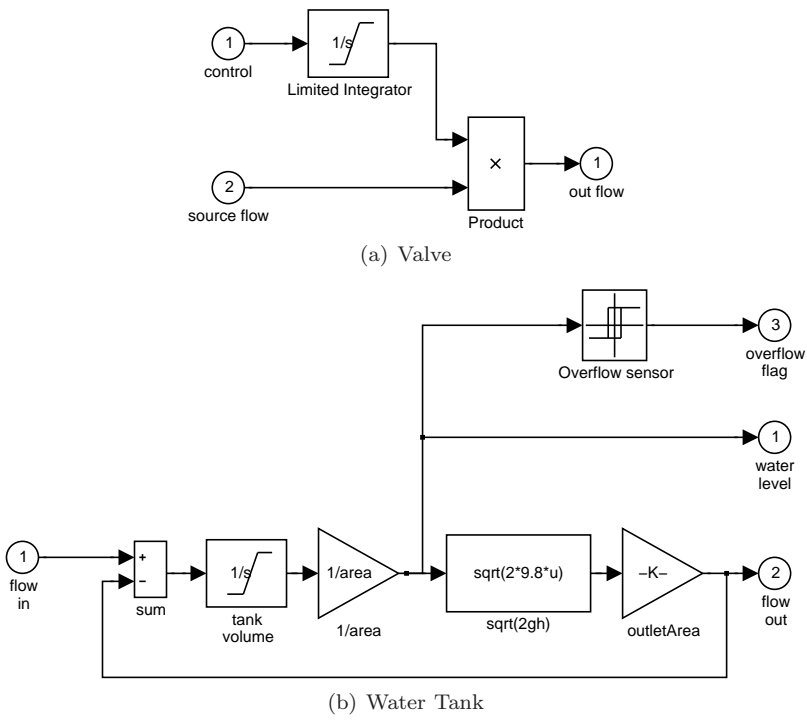


Figure A.6: Subsystems in the water tank demo model.

time for adaptation, a discrete version of the water tank system has been built. In practice this comes down to replacing the (limited) integrators by (limited) discrete time integrators and replacing the derivative by a zero order hold to keep the value of the last evaluation and taking the difference between that value and the current value (see also Figure A.7).

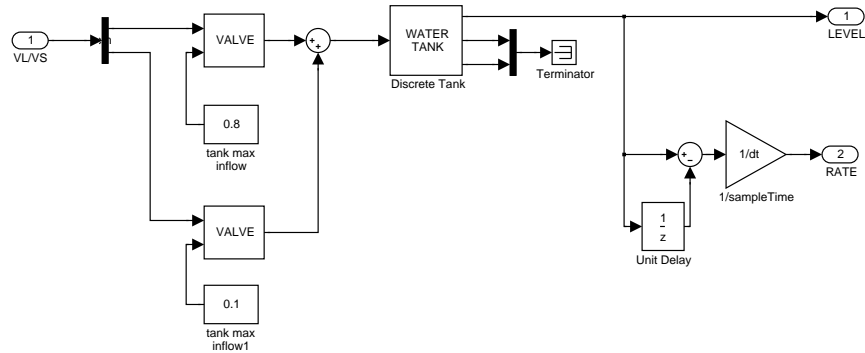


Figure A.7: Discrete SIMULINK implementation of the water tank demo model.

### A.3 Error Messages

This implementation was tested using MATLAB 6.5 (Release 13). Some notes:

- Most simulations should run instantly, but sometimes it may be required to run an initialization script. These scripts are mostly called 'initXXX.m' like 'initFIS.m' for the initialization of a fuzzy inference system. If these do not exist or do not solve the problem, some variables might need a manual initialization, this could be for instance the sample time  $dt = 0.05$ .
- It is possible that during the Adaptive GA optimization process some warnings on sample times occur, these can be neglected. Although the system may seem to have crashed, this is probably not the case: it is just running the adaptation simulation in the background.
- A major problem when using the standard SIMULINK fuzzy logic controller block occurs probably due to a change in the variable types definition in MATLAB between versions 5 and 6. When starting the simulation a pop-up appears with the message:

**Block error - max (COA) Simulink MinMax blocks do not accept 'boolean' signals**

Remedies:

1. Put the FIS evaluation in a .m file (slow, not recommended) or
2. Edit the simulink library:
  - (a) Click the blue link in the error message  
`<modelName>/FuzzyLogicController.../Defuzzification2/Max(COA)`
  - (b) Close the error message dialog
  - (c) Rightclick in the newly opened window
  - (d) choose *Link options rightarrow go to library block*
  - (e) Rightclick the *COA* block and select *look under mask*
  - (f) Go to the *edit* menu and choose *unlock library*
  - (g) Rightclick the *Zero Strength [<=]* and choose the *Relational Operator Parameters* from the menu
  - (h) check *Show additional parameters*
  - (i) choose *specify via dialog* at the *Output Datatype Mode*
  - (j) type: *float('double')*
  - (k) click *ok*
  - (l) choose *file rightarrow save* from the menu

# Appendix B

## Matlab Code

Here some of the basic code is presented. It would take many, many pages to cover all code, but this selection should give a good insight into the program. A full and working version of all tests handled in this report is available on CD.

### B.1 General Code

#### B.1.1 initFIS.m

```
% Initializes the FIS structures for the
% SIMULINK simulation

warning off; % don't print warnings on screen

% initialize all global variables
global fisVL;
global fisVS;
global adaptVL;
global adaptVS;
global settings;
global SPSSwitch;

% initialize the type of setpoint to be used in the simulation
SPSSwitch = 3;

% settings
settings.dt=.05; % timestep [s]
dt = settings.dt;
settings.rangeVL = [-1 1;... % range of FIS inputs
                  -1 1;...
                  ];
settings.rangeVS = [-.3 .3;... % range of FIS inputs
```

```

        -.3 .3;...
    ];
settings.numMF = 5; % number of membershipfunctions

% check ranges
if(size(settings.rangeVL)~=size(settings.rangeVS))
    error(['Ranges are set wrong,' ...
        'All FISes must have the same number of inputs!']);
end

% make the new FISes with numMF MFs with their tops equally
% spaced over the ranges
fisVL = makeFIS(settings.rangeVL,vlinspace(settings.rangeVL(:,1),...
    settings.rangeVL(:,2),settings.numMF));
fisVS = makeFIS(settings.rangeVS,vlinspace(settings.rangeVS(:,1),...
    settings.rangeVS(:,2),settings.numMF));

% set the input range in the FIS itself
% (only used for 'out of range' warnings)
fisVS.input(1).range = [-1 1];
fisVS.input(2).range = [-1 1];

% make FISes based on the fisVL and fisVS
% and add the adaptive rulebase
adaptVL = addAdaptiveRules(fisVL,1); % use the adaptive rulebase
adaptVS = addAdaptiveRules(fisVS,2); % use the adaptive rulebase

% make FISes based on the fisVL and fisVS
% and add the static rulebase
fisVL = addStaticRules(fisVL,1); % use the static rulebase
fisVS = addStaticRules(fisVS,2); % use the static rulebase

```

### B.1.2 makeFIS.m

```

% MAKEFIS makes a fis structure from ranges and the MF-top
% positions. Each input is a row, for instance
% input:  | range:  | tops:
%-----
% 1        | -1  1   | -1 -.5 0 .5 1
% 2        | -1  1   | -1 -.5 0 .5 1
%
% all inputs must have the same number of MFs for they are
% stored in a matrix.

```

```

function fis = makeFIS(range, tops)

% set the appropriate names for the MFs (for 5 or 7 MFs)
if (size(tops,2)==5)
    name = ['NB';'NS';'ZE';'PS';'PB'];
elseif (size(tops,2)==7)
    name = ['NB';'NM';'NS';'ZE';'PS';'PM';'PB'];
else
    % name is just a string of the number
    name = int2str([1:size(tops,2)]');
end

% make the FIS
A=newfis('static');
for i = 1:size(range,1) % loop input variables
    A=addvar(A,'input',int2str(i),range(i,:));
    A=addmf(A,'input',i,name(1,:), 'trapmf',...
        [range(1) range(1) tops(i,1) tops(i,2)]);
    for j = 2:size(tops,2)-1 % loop all MFs excluding the outer most
        A=addmf(A,'input',i,name(j,:), 'trimf',...
            [tops(i,j-1) tops(i,j) tops(i,j+1)]);
    end
    A=addmf(A,'input',i,name(size(tops,2),:), 'trapmf',...
        [tops(i,end-1) tops(i,end) range(i,2) range(i,2)]);
end

% add the output variable and its MFs
A=addvar(A,'output','out',[-1 1]);
A=addmf(A,'output',1,'NB','trimf',[-1 -0.95 -0.9]);
A=addmf(A,'output',1,'NS','trimf',[-.35 -.30 -.25]);
A=addmf(A,'output',1,'ZE','trimf',[-.05 0 .05]);
A=addmf(A,'output',1,'PS','trimf',[.25 .30 .35]);
A=addmf(A,'output',1,'PB','trimf',[0.9 0.95 1]);

fis=A;

```

### B.1.3 addStaticRules.m

```

% function to set the rules of a FIS to match
% the staic rule base. VLVS is an integer
% switch (VLVS = 1 for VL, else for VS)

function fis = addStaticRules(fis,VLVS)
% original ruletable
Rorig=[...textcolorcomment
    %if D= AND DD= THEN VL= and VS= WEIGHT (AND/OR)

```

```

        %NB
        1      1      5      3      1      1      ;...
        2      1      5      3      1      1      ;...
        3      1      5      3      1      1      ;...
        4      1      4      3      1      1      ;...
        5      1      4      3      1      1      ;...
        %NS
        1      2      5      3      1      1      ;...
        2      2      5      3      1      1      ;...
        3      2      4      2      1      1      ;...
        4      2      4      4      1      1      ;...
        5      2      4      2      1      1      ;...
        %ZE
        1      3      5      3      1      1      ;...
        2      3      4      4      1      1      ;...
        3      3      3      3      1      1      ;...
        4      3      2      4      1      1      ;...
        5      3      2      4      1      1      ;...
        %PS
        1      4      4      3      1      1      ;...
        2      4      3      2      1      1      ;...
        3      4      1      3      1      1      ;...
        4      4      1      1      1      1      ;...
        5      4      1      1      1      1      ;...
        %PL
        1      5      3      3      1      1      ;...
        2      5      1      1      1      1      ;...
        3      5      1      3      1      1      ;...
        4      5      1      1      1      1      ;...
        5      5      1      1      1      1      ;...
];

% empty the existing rule base
fis.rule=[];

% add the rules
if VLVS==1
    fis=addrule(fis,Rorig(:,[1,2,3,5,6]));
else
    fis=addrule(fis,Rorig(:,[1,2,4,5,6]));
end

```

#### B.1.4 addAdaptiveRules.m

```

% function to set the rules of a FIS to match
% the adaptive rule base (all 'NB'). VLVS is an

```

```

% integer switch (VLVS = 1 for VL, else for VS)

function fis = addStaticRules(fis,VLVS)

% the adaptive ruletable
Radapt=[...textcolorcomment
        %if D= AND DD= THEN VL= and VS= WEIGHT (AND/OR)
        %NB
        1 1 1 1 1 1 ;...
        2 1 1 1 1 1 ;...
        3 1 1 1 1 1 ;...
        4 1 1 1 1 1 ;...
        5 1 1 1 1 1 ;...
        %NS
        1 2 1 1 1 1 ;...
        2 2 1 1 1 1 ;...
        3 2 1 1 1 1 ;...
        4 2 1 1 1 1 ;...
        5 2 1 1 1 1 ;...
        %ZE
        1 3 1 1 1 1 ;...
        2 3 1 1 1 1 ;...
        3 3 1 1 1 1 ;...
        4 3 1 1 1 1 ;...
        5 3 1 1 1 1 ;...
        %PS
        1 4 1 1 1 1 ;...
        2 4 1 1 1 1 ;...
        3 4 1 1 1 1 ;...
        4 4 1 1 1 1 ;...
        5 4 1 1 1 1 ;...
        %PL
        1 5 1 1 1 1 ;...
        2 5 1 1 1 1 ;...
        3 5 1 1 1 1 ;...
        4 5 1 1 1 1 ;...
        5 5 1 1 1 1 ;...
];

% empty the existing rule base
fis.rule=[];

% add the rules
if VLVS==1
    fis=addrule(fis,Radapt(:,[1,2,3,5,6]));

```

```

else
    fis=addrule(fis,Radapt(:,[1,2,4,5,6]));
end

```

### B.1.5 vlinspace.m

```

function y = vlinspace(d1, d2, n)
%VLINSPACE Linearly spaced matrix.
% VLINSPACE(V1, V2) generates a matrix of 100 linearly
% equally spaced points between column vectors V1 and V2.
%
% Linspace(V1, V2, N) generates N points between V1 and V2.
% For N < 2, Linspace returns V2.
%
% See also Linspace, logspace, :.

if nargin == 2
    n = 100;
end

y = [d1*ones(1,n-1)+(d2-d1)*(0:n-2)/(floor(n)-1) d2];

```

### B.1.6 evalStatic.m

```

% function called by simulink to determine the
% controller output for a static FIS

function [output] = evalAdapt(input)
global fisVL;
global fisVS;

D = input(1); % level
DD=input(2); % rate

% Evaluate the inputs with the adapted FIS
VL = evalfis([D DD],fisVL);

% Evaluate the inputs with the adapted FIS
VS = evalfis([D DD],fisVS);

output=[VL, VS];

```

### B.1.7 evalAdapt.m

```

% function called by simulink to determine the

```

```

% controller output for an adaptive FIS

function [output] = evalAdapt(input)
global adaptVL;
global adaptVS;
global mtrVL; % most triggered rule for VL
global mtrVS; % most triggered rule for VS

D=input(1); % level
DD=input(2); % rate

% if this is not the first call, adapt the rules
if(length(mtrVL)~=0)
    % Adapt consequents of most triggered rule for VL
    if(D<0 && DD<=1e-2) % +1 if not on max
        adaptVL.rule(mtrVL).consequent=...
        adaptVL.rule(mtrVL).consequent...
        +(adaptVL.rule(mtrVL).consequent...
        <length(adaptVL.output(1).mf));
    end
    if(D>0) % -1 if not on min
        adaptVL.rule(mtrVL).consequent=...
        adaptVL.rule(mtrVL).consequent...
        -(adaptVL.rule(mtrVL).consequent>1);
    end
end

% Adapt consequents of most triggered rule for VS
if(D<0 && DD<=1e-2) % +1 if not on max
    adaptVS.rule(mtrVS).consequent=...
    adaptVS.rule(mtrVS).consequent...
    +(adaptVS.rule(mtrVS).consequent...
    <length(adaptVS.output(1).mf));
end
if(D>0) % -1 if not on min
    adaptVS.rule(mtrVS).consequent=...
    adaptVS.rule(mtrVS).consequent...
    -(adaptVS.rule(mtrVS).consequent>1);
end
end

% make sure ZE/ZE ==> ZE/ZE
% (this way is only valid for 5MF controllers)
adaptVL.rule(13).consequent=3;
adaptVS.rule(13).consequent=3;

```

```

% Evaluate the inputs with the adapted FIS
[VL IRR] = evalfis([D DD],adaptVL);
% determine most triggered rule from old inputs
[val mtrVL]=max(sum(IRR'));

% Evaluate the inputs with the adapted FIS
[VS IRR] = evalfis([D DD],adaptVS);
% determine most triggered rule from old inputs
[val mtrVS]=max(sum(IRR'));

output=[VL, VS];

```

## B.2 NN Code

EvalNN.m (§B.2.1) is called from SIMULINK by a ‘matlab function’ block and replaces the ‘Valves+Tank’ block in Figure A.1. Another difference with this figure is that only one (static) controller is implemented, feeding both the official tank and the network simulation with control signals based on state of the official tank.

### B.2.1 evalNN.m

```

function out = evalNN(in)
global ap;
global net;

% to help the NN off
if (isempty(ap))
    % empirically obtained starting values (level;rate)
    ap = [.5;-.1565];
    out = ap;
    return
end

% combine the curent control input and the last level and rate
% to the NN input
inp = [in(1:2);ap(:,end)];
% normalize the input
inp = (inp-net.userdata.offP(1:4)') ./net.userdata.normP(1:4)';
% clip the inputs
inp = min([inp';ones(1,length(inp))]);
inp = max([inp;zeros(1,length(inp))])';
% simulate the network
out = sim(net,inp);
% de-normalize the output
out = out.*net.userdata.normT'+net.userdata.offT';

```

```
ap = [ap out];
```

## B.2.2 makeNN.m

```
% -----
%           ---== !!! WARNING !!! ==---
% -----
% This script will take many many hours to run when
% not edited properly! There are 2 main loops in
% this script:
% - one to train all (selected) networks with
%   datasets of different length
% - one to select the desired networks or
%   network settings.

folder = 'TrainedNets/LevelOnly/'; % folder to store results in

% loop to run it for 3 different dataset lengths
% (type 1:1 to run only for 2500 data-pairs, 2:2 for 5000 only etc.)
for vla = 1:3
    fprintf('\nInitializing data...\n');
    dset = 'MNTrainDataCombined2'; % dataset to use
    arr = [2500, 5000, 10000]; % nr of data pairs in subsequent loops
    n = arr(vla); % no of data pairs to train
    maxn = 250; % max no of neurons
    acc = 2e-6; % accuracy treshold
    % seed=sum(1000*clock);
    seeds = [2134607,2137806,2087650]; % random seeds to use
    seed = seeds(vla);
    rand('state',seed);

    % initialize the data sets
    p=[];
    t=[];
    load(dset);
    p = cat(2,trainData.signals(:).values);
    t = cat(2,trainTargets.signals(:).values);
    % t = t(:,1); % switch on for level only
    % t = t(:,2); % switch on for rate only

    % normalize data
    normP = max(p)-min(p);
    offP = min(p);
    normT = max(t)-min(t);
```

```

offT = min(t);
pn = (p-(ones(length(p),1)*offP))./(ones(length(p),1)*normP);
tn = (t-(ones(length(t),1)*offT))./(ones(length(t),1)*normT);

% use random permutation so no data pair will be taken twice
ind = randperm(length(pn));
ind1 = ind(1:n);
% make training sets and test sets
p1 = pn(ind1,1:4);
t1 = tn(ind1,:);
ind2 = ind(end-n+1:end);
p2 = pn(ind2,1:4);
t2 = tn(ind2,:);
% format matrix properly to be handled by NN
Pseq1 = con2seq(p1');
Tseq1 = con2seq(t1');
Pseq2 = con2seq(p2');

% loop to make all different networks
% (type i = [1 3 5] to only run cases 1,3 and 5 for instance)
for i = [1 3 10 11 12 13 14 17 18 19 20 21 22]
    fprintf('Making network...\n');
    tic;
    warning off;
    switch(i)
        case(1)
            net = newcf([min(pn)' max(pn)'], [10,size(tn,2)],...
                {'tansig' 'purelin'});
        case(2)
            net = newelm([min(pn)' max(pn)'], [10,size(tn,2)],...
                {'tansig' 'purelin'});
        case(3)
            net = newff([min(pn)' max(pn)'], [10,size(tn,2)],...
                {'tansig' 'purelin'});
        case(4)
            net = newgrnn(p1',t1',.75);
        case(5)
            net = newpnn(p1',t1',.75);
        case(6)
            net = newgrnn(p1',t1',.01);
        case(7)
            % does not work properly, gives NaN at simulation
            net = newgrnn(p1',t1',.001);
        case(8)
            % does not work properly, gives NaN at simulation
            net = newgrnn(p1',t1',.0001);
    end
end

```

```
case(9)
    % does not work properly, gives NaN at simulation
    net = newgrnn(pn(ind(1:10000),1:4)',...
        tn(ind(1:10000))',.0001);
case(10)
    net = newcf([min(pn)' max(pn)'],[20,size(tn,2)],...
        {'tansig' 'purelin'});
case(11)
    net = newcf([min(pn)' max(pn)'],[50,size(tn,2)],...
        {'tansig' 'purelin'});
case(12)
    net = newcf([min(pn)' max(pn)'],[10,10,size(tn,2)],...
        {'tansig' 'tansig' 'purelin'});
case(13)
    net = newcf([min(pn)' max(pn)'],[20,10,size(tn,2)],...
        {'tansig' 'tansig' 'purelin'});
case(14)
    net = newff([min(pn)' max(pn)'],[20,size(tn,2)],...
        {'tansig' 'purelin'});
case(15)
    % does not work properly, gives NaN at simulation
    net = newgrnn(pn(:,1:4)',tn(:,:))',.01);
case(16)
    % does not work properly, gives NaN at simulation
    net = newgrnn(pn(:,1:4)',tn(:,:))',.001);
case(17)
    net = newcf([min(pn)' max(pn)'],[5,size(tn,2)],...
        {'tansig' 'purelin'});
case(18)
    net = newcf([min(pn)' max(pn)'],[5,5,size(tn,2)],...
        {'tansig' 'tansig' 'purelin'});
case(19)
    net = newcf([min(pn)' max(pn)'],[25,25,size(tn,2)],...
        {'tansig' 'tansig' 'purelin'});
case(20)
    net = newff([min(pn)' max(pn)'],[50,size(tn,2)],...
        {'tansig' 'purelin'});
case(21)
    net = newff([min(pn)' max(pn)'],[10,10,size(tn,2)],...
        {'tansig' 'tansig' 'purelin'});
case(22)
    net = newff([min(pn)' max(pn)'],[25,25,size(tn,2)],...
        {'tansig' 'tansig' 'purelin'});
case(23)
    net = newelm([min(pn)' max(pn)'],[20,size(tn,2)],...
        {'tansig' 'purelin'});
```

```

        case(24)
            net = newelm([min(pn)' max(pn)'], [50,size(tn,2)],...
                {'tansig' 'purelin'});
        case(25)
            net = newelm([min(pn)' max(pn)'], [10,10,size(tn,2)],...
                {'tansig' 'tansig' 'purelin'});
    end

    % set nr. of training epochs for BP networks and train them
    if(i<4 || (i>=10&&i<15) || i>16)
        net.trainParam.epochs = 50;
        fprintf('Training network...\n');
        net = train(net,Pseq1,Tseq1);
    end

    % store some specification data in the network
    warning on;
    net.userdata.name = ['net' num2str(i) '_' num2str(n)];
    net.userdata.dataSet = dset;
    net.userdata.seed = seed;
    net.userdata.maxNeurons = maxn;
    net.userdata.accuracyTreshold = acc;
    net.userdata.numberofDatapoints = n;
    net.userdata.normP=normP;
    net.userdata.normT=normT;
    net.userdata.offP=offP;
    net.userdata.offT=offT;
    net.userdata.trainingTime=toc;

    % only for nice plots of training errors and test set errors
    % fprintf('Simulating network (with Train Data)...\n');
    % % In case of memory problems
    % tic;
    % for aap = 1:length(p1)
    %     a(:,aap) = sim(net,p1(aap,:));
    % end
    % net.userdata.simnx1Time=toc;
    % result = a';
    % % when no memory problem this can be used
    % % a = sim(net,Pseq1);
    % % result = cat(2,a{:});
    % fprintf('Plotting...\n');
    % figure;
    % plot([result t1]);
    % legend('Sim Level','Sim Rate','Tar Level','Tar Rate');
    % title('Trainset')

```

```

%
%     figure;
%     plot(result-t1);
%     legend('Error in Level','Error in Rate');
%     title('Trainset Error')
%
%     fprintf('Simulating network (with Test Data)...\n');
%     tic;
%     b = sim(net,Pseq2);
%     net.userdata.sim1xnTime=toc;
%     fprintf('Plotting...\n');
%     figure;
%     result = cat(2,b{:})';
%     plot([result t2]);
%     title('Testset')
%     legend('Sim Level','Sim Rate','Tar Level','Tar Rate');
%
%     figure;
%     plot(result-t2);
%     legend('Error in Level','Error in Rate');
%     title('Testset Error')
% -----
% save the network
assignin('base',['net' num2str(i) '_' num2str(n)],net);
save([folder 'net' num2str(i) '_' num2str(n)],'net');
end
end

% compare the networks using test data
compareNN;
% compare the networks using a control simulation
compareNNsim;

```

### B.2.3 compareNN.m

```

% compare the networks using test data

folder = 'TrainedNets/LevelOnly/';
fprintf('\nInitializing data...\n');
dset = 'NNTrainDataCombined2'; %dataset to use
% make test data (copy-paste from makeNN.m)
rand('state',12345);
n=10000;
p=[];
t=[];
load(dset);

```

```
p = cat(2,trainData.signals(:).values);
t = cat(2,trainTargets.signals(:).values);
% t = t(:,1); % switch on for level only
% t = t(:,2); % switch on for rate only
normP = max(p)-min(p);
offP = min(p);
normT = max(t)-min(t);
offT = min(t);
pn = (p-(ones(length(p),1)*offP))./(ones(length(p),1)*normP);
tn = (t-(ones(length(t),1)*offT))./(ones(length(t),1)*normT);

ind = randperm(length(pn));
ind1 = ind(1:n);
p1 = pn(ind1,1:4);
t1 = tn(ind1,:);
ind2 = ind(end-n+1:end);
p2 = pn(ind2,1:4);
t2 = tn(ind2,:);

Pseq1 = con2seq(p1');
Tseq1 = con2seq(t1');
Pseq2 = con2seq(p2');

% find all .mat files in the specified folder
files = what(folder);
nets = files.mat;

res = [];
% loops all networks
for i = 1:length(nets)
    % load the network
    load([folder nets{i}]);

    fprintf('Simulating network (with Test Data)...\\n');
    % evaluate the network
    b = sim(net,Pseq2);
    result = cat(2,b{:})';
    % store the mean test set error
    res = [res ; mean(abs(result-t2))];
end

% save the result
save([folder 'res'], 'res' , 'nets');
```

### B.2.4 compareNNSim.m

```

% compare the networks using a control simulation
global net;
folder = 'TrainedNets/LevelOnly/';

% find all .mat files in the specified folder
files = what(folder);
nets = files.mat;

resSim = [];
% loops all networks
for i = 1:length(nets)
    fprintf('\nSimulating net %d of %d',i,length(nets));
    % if the current mat file is not
    % the file saved by compareNN then ...
    if(strcmp(nets{end},'res.mat')==0)
        % load the network
        load([folder nets{i}]);
        % run the simulink control simulation
        sim('StaticNNDiscr');
    %     sim('StaticNNDiscrLevelOnly');
    % store the mean error
    resSim = [resSim ; mean(abs(simout.signals.values(:,2)...
        -simout.signals.values(:,3))),...
        mean(abs(simout.signals.values(:,4)...
        -simout.signals.values(:,5)))]];
    end
    % -----
end

% save the results to disk
save([folder 'resSim'], 'resSim' , 'nets');

```

## B.3 GA Code

The files used from the third party Genetic Algorithm Optimization Toolbox (GAOT) [22] are not listed here.

### B.3.1 optimizeLong.m

```

%-----
% some lines have been commented to make sure optim10.m works
%-----

% clc;

```

```
% clear;
% clear global;
% close all;
% profile on;
% global fisVL;
% global fisVS;
% global settings;
%
%% relative path to save simulation results
% settings.savepath='opt1/';

% multiplication factor for penalizing overshoot
settings.overshootpenalty = 5;
settings.counter=0; % counts the number of evaluations
settings.dt=0.05; % timestep
settings.ga.popsiz=10; % GA population size
settings.ga.maxgen=25; % GA max number of generations

settings.ga.evalfun = 'fisEval3LongNew';
settings.ga.termfun = 'maxGenTerm';
settings.ga.selfun = 'normGeomSelect';
settings.ga.mutfun = 'nonUnifMutation unifMutation';
settings.ga.xoverfun = 'arithXover simpleXover';

% probability of selecting the best
settings.ga.selParam = [0.06];
% crossover rates
settings.ga.xoverParam = [2;2];
% mutation rates etc.
settings.ga.mutParam = [3 settings.ga.maxgen 1; 2 0 0];
% maximum number of generations
settings.ga.termParam = settings.ga.maxgen;

settings.ga.bounds = [0 1; ...textcolorcomment % bounds for the GA
    0 1; ...
    0 1; ...
    0 1; ...
    0 1; ...
    0 1; ...
    0 1; ...
    0 1; ...
];

settings.rangeVL = [-1 1; ...textcolorcomment % range of FIS inputs
    -1 1; ...
    ];
```

```

settings.rangeVS = [-.3 .3;...textcolorcomment % range of FIS inputs
                  -.3 .3;...
                  ];

if(size(settings.rangeVL)~=size(settings.rangeVS))
    error(['Ranges are set wrong,' ...
          'All FISes must have the same number of inputs!']);
end

settings.numMF = 5; % number of membershipfunctions

% make a new FIS with numMF MFs
fisVL = makeFIS(settings.rangeVL,vlinspace(settings.rangeVL(:,1),...
    settings.rangeVL(:,2),settings.numMF));
fisVL = addStaticRules(fisVL,1); % use the static rulebase
% make a new FIS with 5 MFs
fisVS = makeFIS(settings.rangeVS,vlinspace(settings.rangeVS(:,1),...
    settings.rangeVS(:,2),settings.numMF));
fisVS = addStaticRules(fisVS,2); % use the static rulebase

% initialize GA
fprintf('initializing\n');
% initPop=initializega(settings.popsize,settings.bounds,'fisEval2');
initPop=initializega(settings.ga.popsize,settings.ga.bounds,...
    settings.ga.evalfun);

% optimize
fprintf('optimizing\n');
[x endPop bpop trace] = ga(settings.ga.bounds,...
    settings.ga.evalfun,[],initPop,[1e-4 1 1],...
    settings.ga.termfun,settings.ga.termParam,...
    settings.ga.selfun,settings.ga.selParam,...
    settings.ga.xoverfun,settings.ga.xoverParam,...
    settings.ga.mutfun,settings.ga.mutParam);

% evaluate & display best solution
eval([settings.ga.evalfun '(x,[1])']);
load([settings.savepath 'eval' int2str(settings.counter)]);
save([settings.savepath 'evalBEST'],...
    'fisVL', 'fisVS', 'error', 'simout',...
    'sol','val','trace','settings');

% plot response
figure;
plot(simout.time,simout.signals.values);
title('best controller response');

```

```

xlabel('time');
legend('Contr.Out', 'Level', 'Rate', 'Setpoint')
axis([0 30 -1 2]);

% show MFs
figure;
subplot(2,2,1), plotmf(fisVL, 'input', 1), xlabel('Level VL');
title('Optimized MFs for VL');
subplot(2,2,3), plotmf(fisVL, 'input', 2), xlabel('Rate VL');

subplot(2,2,2), plotmf(fisVS, 'input', 1), xlabel('Level VS');
title('Optimized MFs for VS');
subplot(2,2,4), plotmf(fisVS, 'input', 2), xlabel('Rate VS');

%plot trace
figure;
plot(trace(:,1), trace(:,2), 'b-');
hold on;
plot(trace(:,1), trace(:,3), 'r-');
plot(trace(:,1), trace(:,4), 'g-');
hold off;
xlabel('Generation'); ylabel('Fitness');
legend('Best', 'Avg', 'StDev');

profile off;
profileStatus = profile('status');
profileInfo = profile('info');
save([settings.savepath 'profile'], 'profileStatus', 'profileInfo');

```

### B.3.2 fisEval3LongNew.m

```

% Extensive evaluation function for GA
% - VS and VL membership functions are coupled
% - positive errors have an extra big influence
%   (multiplied by settings.overshootpenalty)
% - fitness consists of (with different scale factors):
%   * integral over the (multiplied) errors
%   * mean settling time
%   * mean overshoot/undershoot
%   * mean positive overshoot
%   * mean negative overshoot
% - uses the evalStaticDiscrNew simulink model (250 second simulation)
function [sol, val] = fisEval2(sol, options)

% make global variables available to this function
global fisVL;

```

```

global fisVS;
global settings;

% remove the old fitness value from the genes
sol=sol(1:end-1);

% check for violation of bounds
if (sum(sol'>settings.ga.bounds(:,2))>0 |...
    sum(sol'<settings.ga.bounds(:,1))>0)
    error('Some solutions are out of bounds');
end

% make the FISes
fisVL = editFIS(fisVL,settings.rangeVL,sol);
fisVS = editFIS(fisVS,settings.rangeVS,sol);

% evaluate the FISes in the simulink model
sim('evalStaticDiscrLongNew');

% determine the error,
% positive errors (overshoot) are considered to be worse
% than negative errors so they are multiplied with a factor
err = settings.overshootpenalty*error.signals.values.*...
    (error.signals.values>0)+error.signals.values;

% take the integral over the error
val=trapz(error.time,abs(err));

ind=0;
% set the first step size to the difference
% between the begin-level and the begin setpoint
step = simout.signals.values(1,5)-simout.signals.values(1,3);
done=0;
count = 0;
% loop over all setpoint changes
while (~done)
    count = count+1;
    % find all indices that have the same setpoint as the first
    % index after the last one of the last setpoint. This means
    % the whole trick will fail if the same setpoint
    % occurs twice!!! (I guess)
    ind = find(simout.signals.values(:,5)==...
        simout.signals.values(ind(end)+1,5));
    % determine the error values on the interval found
    err = simout.signals.values(ind,3)-simout.signals.values(ind,5);
    % determine the stepsize, this will fail for the first step,

```

```

% that's why it was set before and the try/catch was added
try
    step = simout.signals.values(ind(1),5)...
        -simout.signals.values(ind(1)-1,5);
catch
end
% find all indices with an error larger than
% 2 percent of the stepsize
ind2=find(abs(err)>.02*abs(step))+ind(1);
if (isempty(ind2))
    ind2=length(simout.signals.values);
end
% it will return 1 index too high,
% which becomes a problem for the last setpoint
i2 = min(ind2(end),length(simout.signals.values));
% 2 percent settlingtime (index +1 not needed anymore)
ts(count) = simout.time(i2)-simout.time(ind(1));
% calculate overshoot percentage
os(count) = max([sign(step)*err; 0])/step*100;
% check for last setpoint
if (ind(end)==length(simout.signals.values))
    done=1;
end
end

% calculate the performance characteristics
mts=mean(ts); % mean settling time
mos=mean(abs(os)); % mean overshoot
warning off MATLAB:divideByZero; % no warning for an empty set
mpos=mean(os(find(os>0))); % mean of positive overshoots (%)
mnos=mean(os(find(os<0))); % mean of negative overshoots (%)

% warning on for unforeseen stupidities
warning on MATLAB:divideByZero;

% neatenify values if not set properly for a correct calculation
if (isnan(mpos))
    mpos = 0;
end
if (isnan(mnos))
    mnos = 0;
end

% calculate the fitness value
val = val +.05*mts +.2*mos +mpos +.1*abs(mnos);
val=-val; % GA is maximizing

```

```

% set the function output
sol = [sol val];

% count this evaluation and eventually save it
% (not all are saved because a lack of disk space)
settings.counter = settings.counter+1;
if (mod(settings.counter,10)==0 || ...
    (~isempty(options) && options(1)==1))
    save([settings.savepath 'eval' int2str(settings.counter)], ...
        'fisVL', 'fisVS', 'error', 'simout', 'sol', 'val', 'settings');
end
fprintf('counter = %d \n', settings.counter);

```

### B.3.3 editFIS.m

```

% EDITFIS(f,range,sol) edits the MFs of an existing fis to make it
% consistent with the sol.
%
%   f - the fis to edit
%   range - the ranges of the fis-inputs
%   sol - the solution to set
%
% Returns the edited fis
%
% Sol must have must be of length numInputs*(numMFs-1) and all
% values in sol must be between 0 and 1.
%
% Sol is split into parts for each input. The top of the center MF
% is set at 0. For 5 MFs the other tops are set as following:
% - top of MF2 = solPart(2)*(min range)
% - top of MF1 = solPart(1)*(solPart(2)*(min range)-min range)
% - top of MF4 = solPart(3)*(max range)
% - top of MF5 = solPart(4)*(solPart(3)*(max range)-max range)
%
% All inputs must have the same number of MFs for they are stored
% in a matrix.

function f = editFIS(f, range, sol)

nin = size(range,1); % number of FIS inputs
a=length(sol)/nin; % number genes for the MFs (numMF-1)

% central MF always has its top at 0, we want an odd number
% of MFs so a must be even
if(mod(a,2)~=0)

```

```

    error('Number of genes for each input must be even!');
end

rsol=reshape(sol,a,nin)';
% calculate positions of tops
tops=zeros(nin,a+1);
tops(:,a/2+1)=(range(:,1)+range(:,2))/2;
for i=1:a/2
    tops(:,a/2+1-i)=tops(:,a/2+2-i)-...
        (tops(:,a/2+2-i)-range(:,1)) .* rsol(:,a/2+1-i); % to left
    tops(:,a/2+1+i)=tops(:,a/2+i)+...
        (-tops(:,a/2+i)+range(:,2)) .* rsol(:,a/2+i); % to right
end

% edit the fis
for i = 1:size(range,1)
    f.input(i).mf(1).params = ...
        [range(1) range(1) tops(i,1) tops(i,2)];
    for j = 2:size(tops,2)-1
        f.input(i).mf(j).params = ...
            [tops(i,j-1) tops(i,j) tops(i,j+1)];
    end
    f.input(i).mf(size(tops,2)).params = ...
        [tops(i,end-1) tops(i,end) range(i,2) range(i,2)];
end

```