ISSN 0280–5316
ISRN LUTFD2/TFRT--5643--SE

Linear and Neuro Control Strategies; Some Experimental Results

Henrik Olsson

Department of Automatic Control
Lund Institute of Technology
June 2000

Department of Automatic Control
Lund Institute of Technology
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
June 2000

Document Number
ISRN LUTFD2/TFRT--5643--SE

Author(s)
Henrik Olsson

Supervisor
Per Hagander, LTH and Antonio Dourado, Coimbra

Sponsoring organisation

Title and subtitle
Linear and Neuro Control Strategies; Some Experimental Results. (Linjära och neurala reglerstrategier; Några experimentella resultat.)

Abstract

The variety of different controllers is today very big. Much research is done in the field of neural network control and more well established technologies are sometimes looked at as old fashioned. The work is much based on the identification of a process model.

In this report some different controllers have been built to see if neural networks are a good alternative to classic controllers. The classical controllers are a pure linear controller and an adaptive linear controller working with updating by a covariance matrix. The neural network controller is based on a process model made by a recurrent neural network. This model is used in an input-output feedback linearisation controller.

The identification with the neural network model is very difficult to make. As often in the world of control the theory by far exceeds reality. It is possible to build good neural network controllers but they are today more another way of achieving the same thing as the conventional controllers.

Key words

Classification system and/ or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
82

Recipient's notes

Security classification

Table of content

1. Introduction

1.1. Overview of the work

The work consists in the construction of different kinds of controllers and to find out their potentials and limitations.

The linear regulator is the most basic in the construction and also the oldest. This controller is therefore good as a reference. (Åström and Wittenmark, 1997) is the by far most used source for the construction of the linear controller.

In the case of adaptive controllers the studies are oriented towards different ways of updating the covariance matrix. A pole-placement control method gives the desired control behaviour. These are well-established technologies on which many books have been written. In the studies for this project are (Landau *et al*, 1997), (Wellstead and Zarrop, 1991) and (Åström and Wittenmark, 1995) the most read.

The neural network is used to create a nonlinear model of the process. This model is used to build an input-output feedback linearisation controller. This approach was first presented by (Isidori, 1995). To see if this new technology has any great advantages in comparison to the older adaptive approaches is the most important conclusion in this report. The material used for this part is mainly (Braake *et al*, 1997), (Henriques *et al*, *not yet published*), (Mathworks, 1999) and (de Wilde, 1997).

1.1.1. Classic Control: Advantages and disadvantages

With classic control we normally mean linear control. In this project the meaning also include the word discrete. A computer makes all the calculations and because of this we deal with discrete time. This is the first method used for this project but creates the foundation for all other control methods. Linear discrete control is based on the assumptions that the process can be modelled as linear and constant polynomials. It is relatively simple to construct a linear controller and the calculations are simple. This means that we can sample with a high speed. The disadvantages are that in real life you cannot always say that a linear and constant model will be good enough. The fact that the behaviour of the process is changing within time, due to changed conditions in or in the nearby environment of the process, can make the system run with wrong behaviour and if you are unlucky, make it unstable. In this project these nonlinearities and variations in parameters will be compensated with adaptive control methods.

1.1.2. Adaptive control: Try to overcome some difficulties of the conventional control

Adaptive control is an attempt to adjust the elements of the model polynomials at the same time as you run the controller. By predicting the next output with regards of the model and then evaluate the difference between the prediction and the actual output it is possible to modify the parameters of the model to the better. There are many different ways of doing this. Only methods including covariance management are used in this report.

1.1.3. Neural networks control: Able to deal with nonlinearities

A linear model cannot describe all processes. As always there are various approaches to control a nonlinear process and one group of methods are the ones that use neural networks. Neural networks can for example be used to create an arbitrary transfer function, linear or nonlinear. The input-output feedback linearisation is one of the possible approaches in order to design a controller. By a linearisation of the neural network model at a working point and at each sample, it is possible to control the system as if it was a linear process.

1.2. Equipment

1.2.1. The PT326 process

The process used for this work was a single input single output process called PT326 from the British manufacturer Feedback (see Figure 1-1).

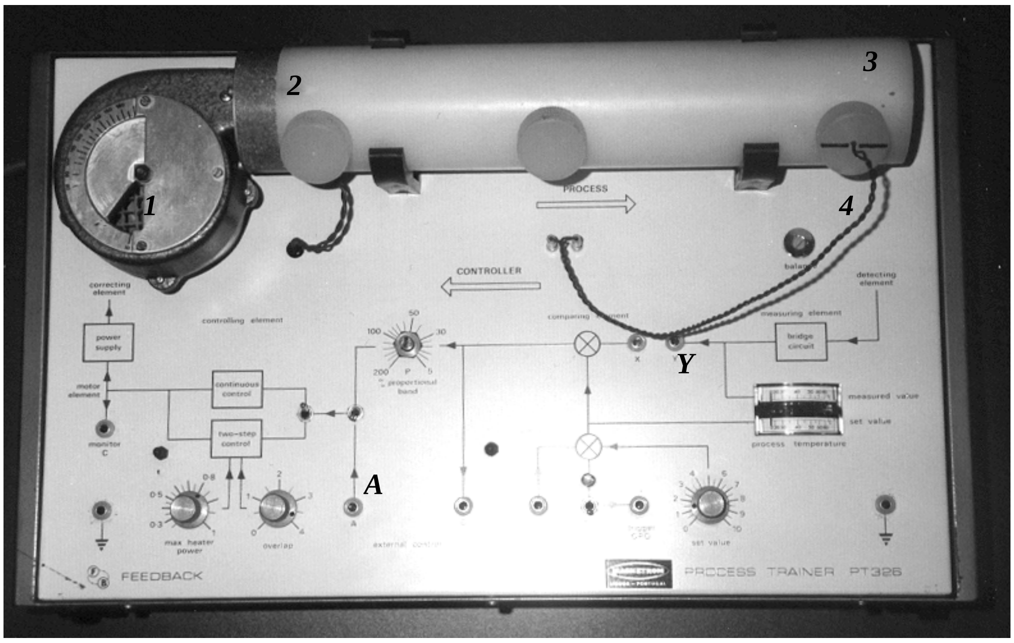


Figure 1-1 *The PT326 process*

Air is sucked in from the ambient atmosphere through an adjustable entrance (1) and driven through an electrical heater grid (2) and then through a plastic tube out in the atmosphere again (3). It has many similarities with a hair dryer. The control problem is to control the temperature of the outgoing air. The input signal (socket A) is a voltage that produces a current through the heater grid and the output is a voltage that is a measurement of the temperature of the outgoing air.

The detecting element is a bead thermistor that is fitted at the end of a probe (4). The probe can be placed in three different positions along the tube, which causes different time delays. The distances are 1.1 inch (28mm), 5.5 inches (140mm) and 11 inches (279mm) from the heater grid (Feedback, 1986).

The thermistor forms one arm of a D.C. bridge that is in balance at 40°C. The output voltage from the bridge is amplified so a change in temperature from 30° to 60° equals a change in output voltage from 0V to +10V. This output voltage is measured at the socket Y on the front panel (Feedback, 1986).

To illustrate disturbances it is possible to adjust the size of the inlet. This changes the speed of the air flow. To measure this disturbance it is possible to read how many degrees the opening is, between 10º and 170º.

1.2.2. Acquisition board PCL-818L

The communication between the process and the computer was handled by acquisition card PCL-818L from Advantech Co, Ltd. PLC-818L contains a 12-bit A/D converter with sampling rates up to 40kHz and a 12-bit D/A converter with 5 microseconds settling time. The range of the D/A converter is ±10V (Advantech, 1994).

1.2.3. Structure of the software

The programming in this work was made in a slightly different way than usual. Many of the functions were written in Matlab. The reason for that is that many functions are very easy to implement and there are many very powerful commands available in Matlab. One example is to invert a matrix, another calculations with complex numbers. These are very complex operations in normal programming languages but are very simple to handle in Matlab. Matlab does not support any possibilities to use the data acquisition card but provides a compiler toolbox that makes it possible to convert the Matlab code to C-code. With the help of this compiler toolbox as many functions as possible were written in Matlab code and then compiled to C-code. Some additional c-functions were thereafter compiled together with the code Matlab had created. A schematic diagram is shown in Figure 1-2. The C-code makes the calculations faster than Matlab, even though the compiler does not make the code as fast as a skilled programmer would make it.

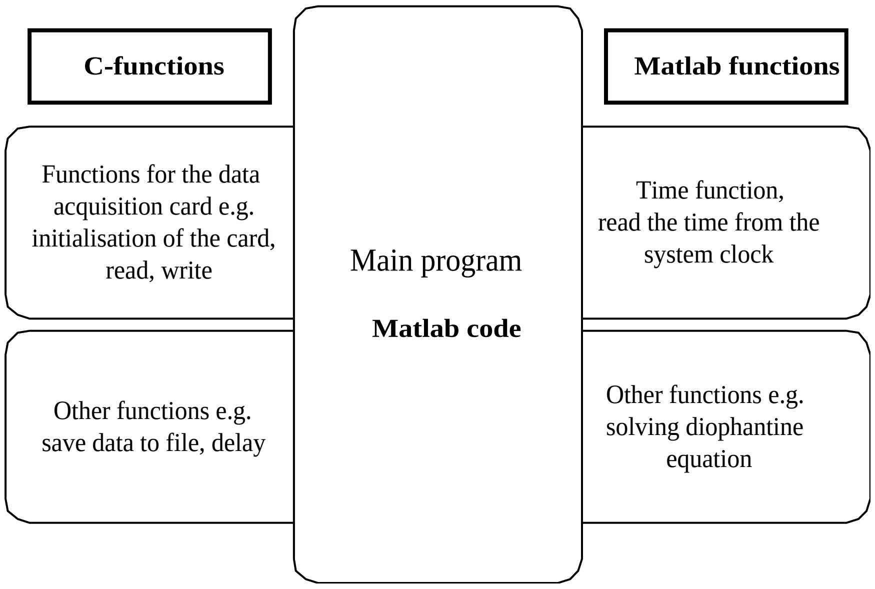
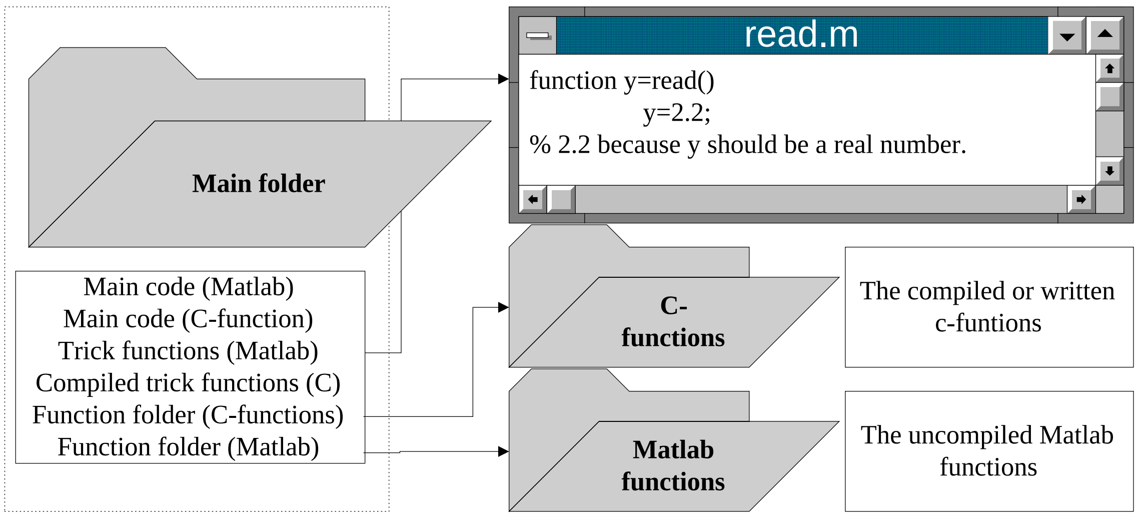

C-functions
Matlab functions
Functions for the data acquisition card e.g. initialisation of the card, read, write
Main program
Matlab code
Time function, read the time from the system clock
Other functions e.g. save data to file, delay
Other functions e.g. solving diophantine equation

Figure 1-2 *This is what language was used for the different functions*

To learn how the Matlab compiler works is very difficult but when you have understood exactly how the structure should be it is not a problem. All Matlab files, even the main program, should start with the word *function*. All functions should be compiled with the –r and –e flags. When the main program is compiled the –m flag should be added too. For every C-function (including the compiled Matlab functions) that is used there should be a short Matlab function attached in the same catalogue as the main program. This function should contain information on what kinds of variables are returned to the main program, e.g. real, integer etc. These functions are in this report called *trick functions* because they trick the compiler to separate name of functions from names of variables in the main program. See also Figure 1-3. These trick functions should be compiled as normal Matlab functions before the compilation of the main code.



```
function y=read()
          y=2.2;
% 2.2 because y should be a real number.
```

Figure 1-3 *One way of arranging the files when working with the Matlab compiler*

The c-code is then compiled as usual with a normal C-compiler. Here was Watcom IDE 10.5 used.

2. Conventional control theory

2.1. The model

The model of the process is described as two polynomials *$A(q^{-1})$* and *$B(q^{-1})$* of the backward-shift operator *q^{-1}*. Sometimes this operator is called the delay operator. The z-transform of these polynomials are: *$A(z^{-1})$* and *$B(z^{-1})$*.

$A(q^{-1}) = 1 + a_1 q^{-1} + a_2 q^{-2} + \dots + a_n q^{-n}$ and $B(q^{-1}) = b_0 + b_1 q^{-1} + \dots + b_m q^{-m}$ so that

$$\begin{aligned} &y(k) + a_1 y(k-1) + a_2 y(k-2) + \dots + a_n y(k-n) = \\ &= b_0 u(k) + b_1 u(k-1) + \dots + b_m u(k-m) + e(k) \end{aligned} \qquad \textbf{2-1}$$

where *$e(k)$* is white noise.

Another way to formulate the equation is: $y = \frac{B(z^{-1})}{A(z^{-1})} u + \frac{1}{A(z^{-1})} e$. This type of model is called the ARX-model. Another type is the ARMAX-model that includes a more detailed description of the noise.

2.2. The controller

2.2.1. Pole placement controller

A linear pole placement control system can be illustrated as below in Figure 2-1:

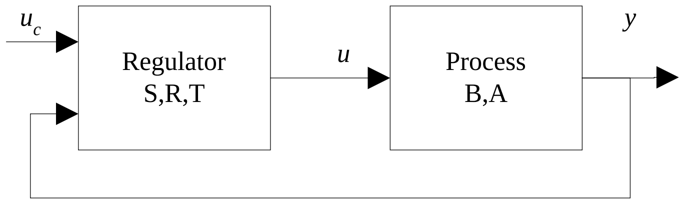


Figure 2-1 *A linear control system*

The output of the process is a function of the control signal, *u(k). u(k)* is at the same time the output of the controller. The inputs to the controller are the old process outputs and the reference signal $u_c(k)$. A linear controller can then be represented as:

$$R(q^{-1})u(k)=T(q^{-1})u_c(k)-S(q^{-1})y(k) \qquad \textbf{2-2}$$

where the linear polynomials can be described as: $R\left(z^{-1}\right)=1+r_1z^{-1}+r_2z^{-2}+\ldots+r_nz^{-n}$, $T\left(z^{-1}\right)=t_0+t_1z^{-1}+t_2z^{-2}+\ldots+t_tz^{-t}$ and $S\left(z^{-1}\right)=s_0+s_1z^{-1}+s_2z^{-2}+\ldots+s_mz^{-m}$. After a division by $R(z^{-1})$ we get:

$$u=\frac{T(z^{-1})}{R(z^{-1})}u_c-\frac{S(z^{-1})}{R(z^{-1})}y\,.$$

If we combine the two equations the output, *y(k)* will be a function of only the reference, $y(k)=f\left(u_c(k)\right)$.

$$\frac{R(z^{-1})A(z^{-1})}{B(z^{-1})}y=T(z^{-1})u_c-S(z^{-1})y\Rightarrow R(z^{-1})A(z^{-1})y=BT(z^{-1})u_c-S(z^{-1})B(z^{-1})y\Rightarrow$$
$$B(z^{-1})T(z^{-1})u_c=\left[A(z^{-1})R(z^{-1})+B(z^{-1})S(z^{-1})\right]y$$

By dividing with [$A(z^{-1})R(z^{-1})+B(z^{-1})S(z^{-1})$] we get the transfer function:

$$y=\frac{B(z^{-1})T(z^{-1})}{A(z^{-1})R(z^{-1})+B(z^{-1})S(z^{-1})}u_c \qquad \textbf{2-3}$$

Both $B(q^{-1})T(q^{-1})$ and $A(q^{-1})R(q^{-1})+B(q^{-1})S(q^{-1})$ are linear equations and we can give them new names in analogy with the transfer function of the process:

$$B_c(q^{-1})=B(q^{-1})T(q^{-1}) \text{ and } A_c(q^{-1})=A(q^{-1})R(q^{-1})+B(q^{-1})S(q^{-1})$$

The latter equation is called the Diophantine equation.

2.2.2. Calculating the regulator polynomials

The control problem is to design and implement the controller polynomials so the behaviour of the system is as good as possible. First of all you should try to place the poles of A_c where you want them. By having a desired A_c you identify the terms of the resulting polynomials of the Diophantine equation. That is easily done with the following equation system that is valid in the general case (Åström and Wittenmark, 1997).

$$\begin{bmatrix} 1 & 0 & \dots & 0 & b_0 & 0 & \cdots & 0 \\ a_1 & 1 & \ddots & \vdots & b_1 & b_0 & \ddots & \vdots \\ a_2 & a_1 & \ddots & 0 & b_2 & b_1 & \ddots & 0 \\ \vdots & \vdots & \ddots & 1 & \vdots & \vdots & \ddots & b_0 \\ a_n & \vdots & & a_1 & b_n & \vdots & & b_1 \\ 0 & a_n & & \vdots & 0 & b_n & & \vdots \\ \vdots & \ddots & \ddots & & \vdots & \ddots & \ddots & \\ 0 & \cdots & 0 & a_n & 0 & \cdots & 0 & b_n \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_{n-1} \\ s_0 \\ \vdots \\ s_{n-1} \end{bmatrix} = \begin{bmatrix} a_{c1} - a_1 \\ \vdots \\ a_{cn} - a_n \\ a_{cn+1} \\ \vdots \\ a_{c\,2n-1} \end{bmatrix}$$

The degrees of the polynomials *$A(q^{-1})$* and *$B(q^{-1})$* are *n*. If they do not have the same length it is possible to add zeros at the beginning of *$B(q^{-1})$* until they have the same length. The degree of *$R(q^{-1})$* and *$S(q^{-1})$* is *n-1*.

To add an integrator to the regulator you need a regulator pole in z=1. The polynomials *$R(q^{-1})$* and *$S(q^{-1})$* has then the degree *n*. This means that the Diophantine equation is changed slightly:

$$\begin{bmatrix} 1 & 0 & \cdots & 0 & b_0 & 0 & \cdots & 0 \\ a_1 & 1 & \ddots & \vdots & b_1 & b_0 & \ddots & \vdots \\ \vdots & a_1 & \ddots & 0 & \vdots & b_1 & \ddots & 0 \\ a_n & & \ddots & 1 & b_n & \vdots & \ddots & b_0 \\ 0 & a_n & & a_1 & 0 & b_1 & & b_1 \\ \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_n & \vdots & & \ddots & b_n \\ 1 & \cdots & \cdots & 1 & 0 & \cdots & \cdots & 0 \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \\ s_0 \\ \vdots \\ s_n \end{bmatrix} = \begin{bmatrix} a_{c1} - a_1 \\ \vdots \\ a_{cn} - a_n \\ a_{cn+1} \\ \vdots \\ a_{c\,2n} \\ -1 \end{bmatrix}$$

With *$T(q^{-1})$* you can decide the steady state gain which can be calculated as:

$$sum(B_c) / sum(A_c)$$

To force the steady state gain to be a certain value you need a *T(q^{-1})* that is just a constant t_0. *T(q^{-1})* is also used to place the zeros of the system where you want them. In this report these methods are not discussed, but the *T(q^{-1})* might be of a higher order. Further on we make the assumption that *T(q^{-1})=t_0*.

2.2.3. Calculating the control signal

Now we have the regulator. The control signal is now computed as:

$$\begin{aligned} R(q^{-1})u = T(q^{-1})u_c - S(q^{-1})y \Rightarrow u(k) = t_0u_c(k) - s_oy(k) - s_1y(k-1) - \ldots \\ \ldots - s_py(k-p) - r_1u(k-1) - r_2u(k-2) - \ldots - r_mu(k-m) \end{aligned} \quad \textbf{2-4}$$

where *p* and *m* are the degrees of the *S(q^{-1})* and *R(q^{-1})* polynomials respectively.

That was a short version of the theory behind the linear control. There are other ways to obtain the same results but they provide more or less the same result.

2.3. Off line identification

To be able to have a good controller you need to know the behaviour of the process. This means in mathematical terms that you should know the parameters of *A(q^{-1})* and *B(q^{-1})*.

There are two general methods to decide these polynomials. One is based on time-series that are sent to the input of the process. Then the outputs are measured and from these data the parameters are calculated. By measure the output you can calculate the system parameters. The other method is based on calculation of the physical behaviour of the process, e.g. the calculation of the behaviour of an electric net from the resistance, capacitance and impedance of the different components. This includes often very complex calculations and is seldom done. In this report only the first method is used.

By sending an input signal *u(k),* with the length *n,* to the input of the process you could theoretically calculate an exact model for the specific input, if the length of the polynomials are *n*. This model is very complex and is only valid for that specific data. If the output had been measured again it would be slightly different and the model would not be correct. If the assumption is made that *B(q^{-1})* and *A(q^{-1})* can be approximated as polynomials with a length that is much smaller than *n*, the problem is to choose these parameters as good as possible.

The approximated model follows the equation:

$$A(q^{-1})y \approx B(q^{-1})u \qquad \textbf{2-5}$$

as good as possible.

There are many different methods to find the best approximation but the least-mean-square method is very much used, and is the method for this report. This method is going to be presented here.

From the linear model we get that:

$$\begin{aligned}
&y(k) \approx \hat{b}_0 u(k) + \hat{b}_1 u(k-1) + \ldots + \hat{b}_n u(k-n) - \hat{a}_1 y(k-1) - \hat{a}_2 y(k-2) - \ldots - \hat{a}_n y(k-n) = \\
&\left(u(k) \quad u(k-1) \quad \cdots \quad u(k-n+1) \quad -y(k-1) \quad -y(k-2) \quad \cdots \quad -y(k-n)\right) * \\
&* \left(\hat{b}_1 \quad \hat{b}_2 \quad \cdots \quad \hat{b}_n \quad \hat{a}_1 \quad \hat{a}_2 \quad \cdots \quad \hat{a}_n\right)^T = \phi^T(k)\hat{\Theta}
\end{aligned}$$

where $\phi^T(k)$ is a vector containing the *n* latest control signals and outputs and $\hat{\Theta}$ is a vector that contains the estimated model parameters.

Let us also introduce the matrices

$$\Phi_N = \begin{pmatrix} \phi_1^T \\ \phi_2^T \\ \vdots \\ \phi_N^T \end{pmatrix}, \; Y_N = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \text{ and } \varepsilon_N(\overline{\Theta}) = \begin{pmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_N \end{pmatrix} = Y_N - \Phi_N \overline{\Theta} \qquad \textbf{2-6}$$

where $\varepsilon_N(\overline{\Theta})$ is a vector containing the errors when an arbitrary estimation $\overline{\Theta}$ has been used.

If we introduce a loss function $J(\overline{\Theta})$ and define it as:

$$J(\overline{\Theta}) = \sum \left| y(i) - \phi^T(i)\overline{\Theta} \right|^2 = \frac{1}{2}\sum_{k=1}^{N} \varepsilon_k^2 = \frac{1}{2}(Y_N - \Phi_N\overline{\Theta})^T (Y_N - \Phi_N\overline{\Theta}). \qquad \textbf{2-7}$$

For the least-square method we get a minimal loss when

$$\hat{\Theta} = (\Phi_N^T \Phi_N)^{-1} \Phi_N^T Y_N \,. \qquad \textbf{2-8}$$

The proof is to be read in a book in identification, for example (Johansson, 1993) or (Landau, 1990).

3. Experimental results with the fixed linear control

3.1. Modelling

There are many things to consider when you choose the sampling time. First of all you got to know how fast the process is. The PT326 process has a rise time about two seconds. A rule of thumb is to choose the sampling time 4-10 times faster than the rise time. For the PT326 this means a sampling time between 200 and 500ms. A lower sampling rate will not reconstruct the continuous signal properly and a higher sampling rate might increase the load of the computer. Another problem is the noise. If the sampling rate is too high there is a risk that the model you get is a description of the noise instead of the behaviour of the actual process you want to control. With a sampling time big enough the covariance for the noise is approximately 0, and can therefore considered be white. White noise does not affect the identification. With a covariance of the noise separated from 0, the colour of the noise will affect the identification in a negative way. The output from the PT326 process contains a lot of noise and every attempt to make the identification with a sampling time faster than 150ms ended up in very bad models. At the end 200ms seemed to be a reasonable sampling rate. That is the sampling rate that is used for all experiments in this report.

There are different ways of choosing the input signal for the identification. If the process is linear we do not have to use more than two different input levels. It is more useful to vary the step length at each period and keep the step amplitude constant. This lead to the conclusion that it might be useful to choose a rectangular signal with variable step length. This kind of signal is usually called *pseudorandom binary sequence* (PRBS) (see Figure 3-1). This kind of signal is usually a good choice because its frequency spectrum is close to the frequency spectrum of white noise. There are many other ways to determine the input but this is a signal often used (Johansson, 1993).

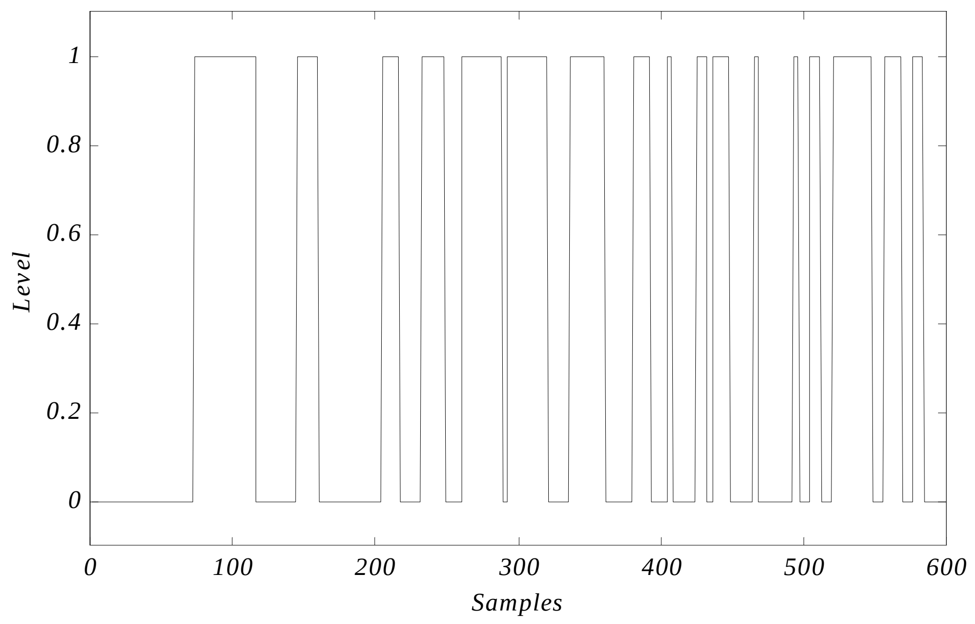


Figure 3-1 *An example of a PRBS signal*

One thing you should have in mind is to make sure that the sampling time really is accurate. In the first experiments was a clock built by C-functions used. The operative system of the computer was Windows98. Windows98 is a multitasking operative system and the C-functions used were only counting the time used for the data acquisition program. In between the computer was working with things only known by the people at Microsoft. This leads of course to errors in the model. Therefore it was more convenient to use the clock function in Matlab for this purpose.

Because of the noise in the signal it might be useful to filter the signal with a low-pass filter.

The order of the model is important. The complexity of the calculations is increasing very fast with increasing order. A simulation might be made to see how well the model is reconstructing the output with the same input. It is important to choose the model with the lowest complexity possible. If the process is a second order system a third order model does not provide a better description, though it is more probable that the noise will be more noticeable in the model.

The results from the identification can be seen in Figure 3-2. The figure also contains a simulation of the model.

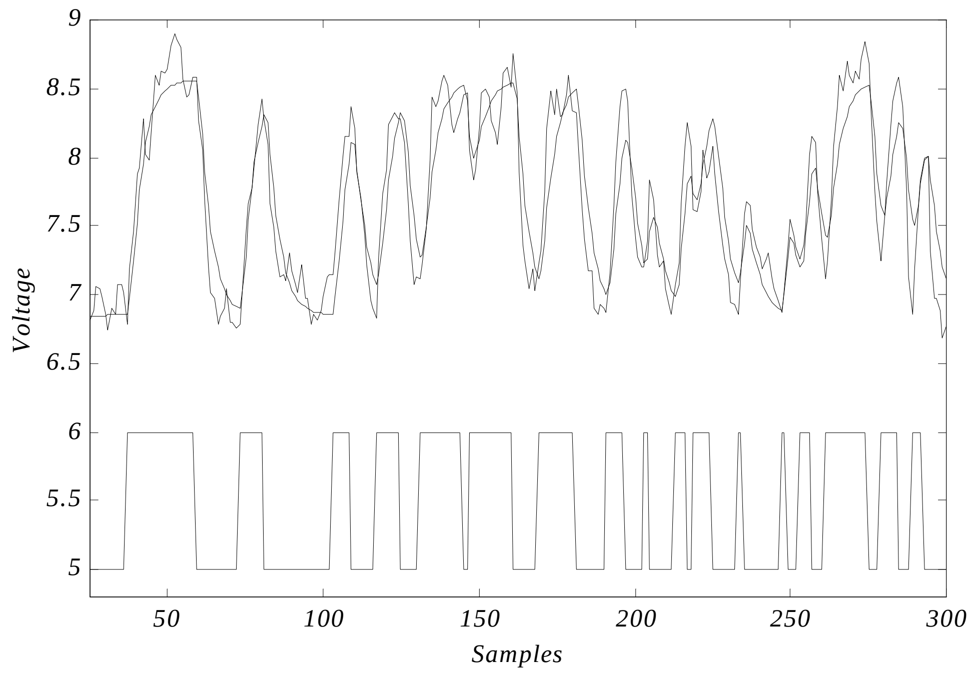


Figure 3-2 *The result of the linear identification. The insignal can be viewed at the bottom and the process output and the simulated output (smoother) can be viewed above.*

The picture shows a fairly good model of the PT326. This model is used for the linear control. The transfer function of the model is:

$$H(z) = \frac{B(z^{-1})}{A(z^{-1})} = \frac{0.1249z^{-1} + 0.1264z^{-2}}{1 - 1.1628z^{-1} + 0.3095z^{-2}}. \qquad \textbf{3-1}$$

The number of the delays is one sample (200ms).

The PRBS is only changing between two levels. If the process is has nonlinearities the model might be quite bad. If the input signal is varied over a wide range within the dynamic range the model errors due to the nonlinearities are also least mean square approximated. The result from the identification with the other insignal can be seen in Figure 3-3 and the transfer function became:

$$H\left(z^{-1}\right) = \frac{B\left(z^{-1}\right)}{A\left(z^{-1}\right)} = \frac{-0.0018z^{-1} + 0.0362z^{-2}}{1 - 1.6990z^{-1} + 0.7257z^{-2}} \qquad \textbf{3-2}$$

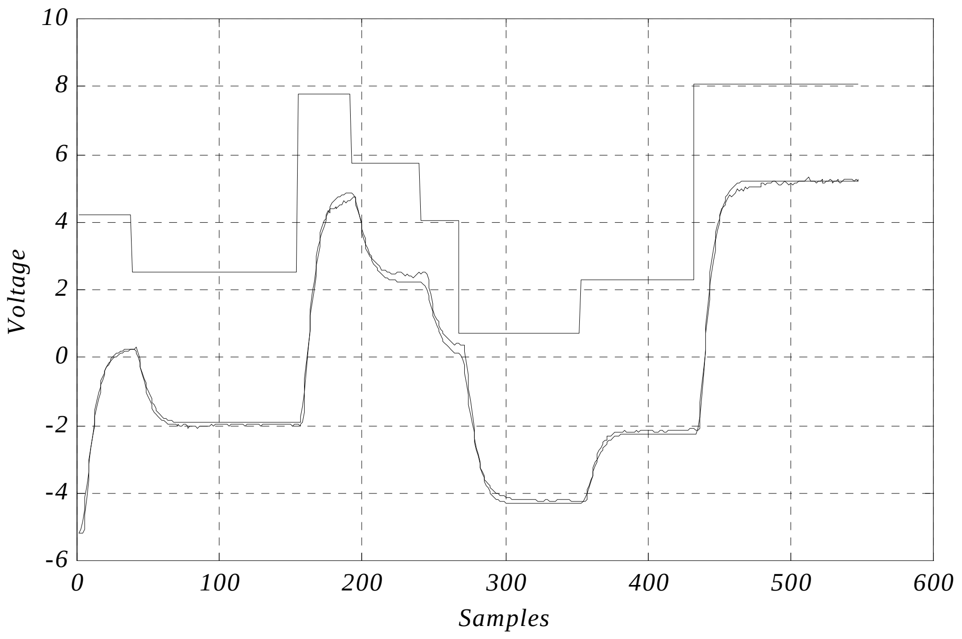


Figure 3-3 *The result of the linear identification. The input can be seen above and the process output and simulated output (smoother) can be viewed below.*

If the model from the PRBS was simulated with the latter insignal the result was the following (see Figure 3-4):

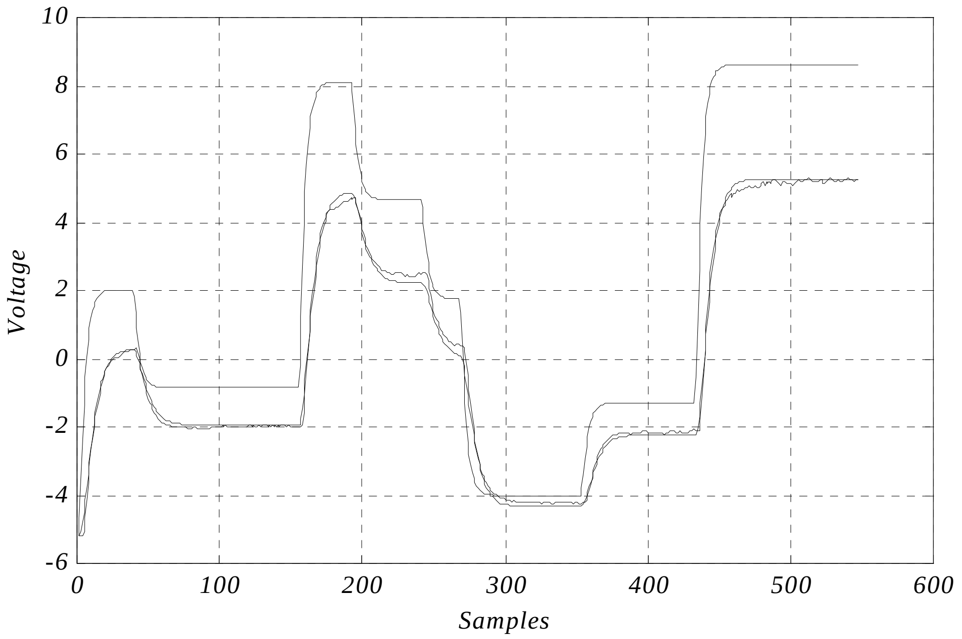


Figure 3-4 *The result of the comparison between the two different models. The model from the PRBS input is the highest one while the real output and the model based on an input with variable amplitude can be seen below.*

The difference is big. It is easy to believe that the second model is the better one but if we make the same simulation on the PRBS input the result is the following (see Figure 3-5):

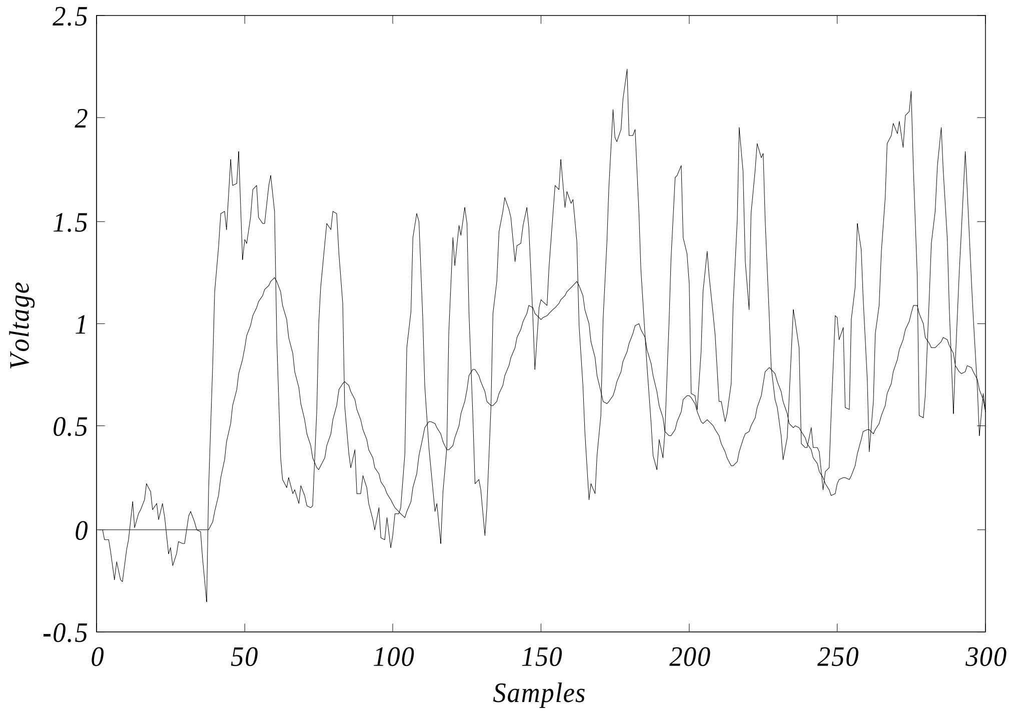


Figure 3-5 *The result of the comparison between the two different models. At the top of the graph is the real output to be seen while further down is the simulated output from the model based on the input with variable amplitude.*

This model is not as good as the first one used. This is due to the small terms of the B-polynomial. The output is almost independent of the insignal. This illustrates how difficult it is to get a perfect model.

3.2. Control

In real life control the noise can cause a lot of trouble. One way to solve this is to filter the output signal before it enters the regulator. It is also possible to filter the control signal. Because the PT326 is noisy, both those filters, that can bee seen in Figure 3-6 are used to control the process.

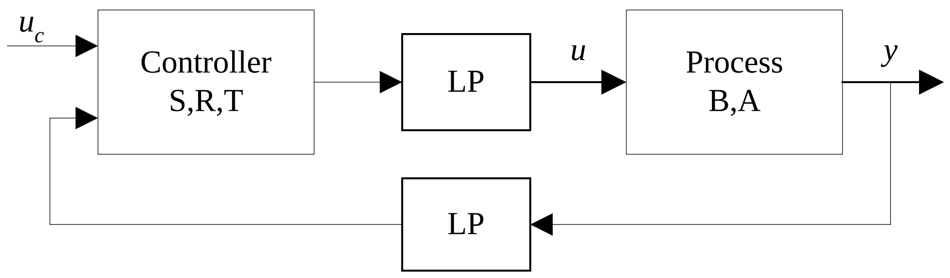


Figure 3-6 *When the output and the control signal is low-pass filtered a good control is much easier to achieve.*

The filters are removing some of the noise but there is still a lot left, which might cause problems.

It might be difficult to find the optimal place to put the closed loop poles. One way is trial and error. Experience is one of the most powerful tools when it comes to find a good location for the poles. To put them around 0.7 on the real axis is often a good start. In the experiments were the poles placed in 0.65 and 0.7. The Diophantine equation gives: $R(q^{-1})=1-0.0520q^{-1}-0.9480q^{-2}$, $S=15,3308-10.8872q^{-1}+2.3212q^{-2}$ and $T=5.3721$.

The model is never a really good approximation of the process. For example the steady state error might be rather big. To prevent this kind of errors it is a good idea to implement an integrator in the regulator.

With all the methods mentioned above including an integrator the control was as can be seen in Figure 3-7.

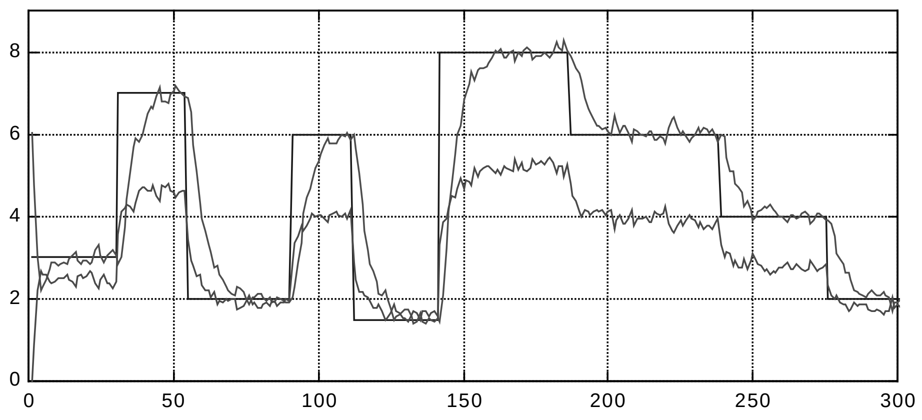


Figure 3-7 *The result with linear control and integrator. The reference (rectancular), the output and the control signal (below the output)*

This control was made straight after the identification was made. When the same model was used some days later, when it was colder, the result was much different (see Figure 3-8)

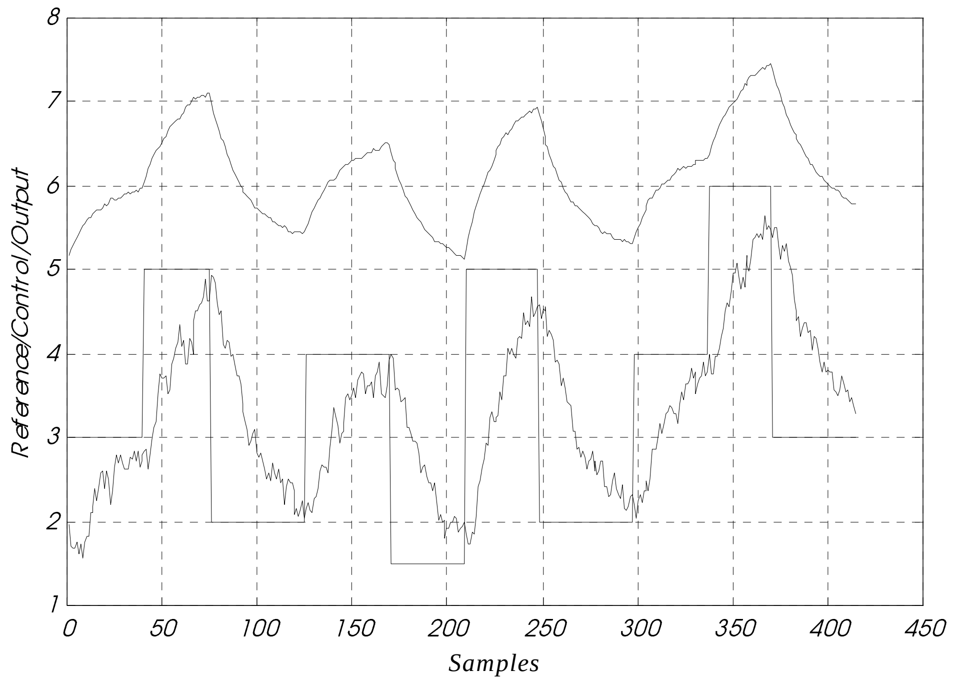


Figure 3-8 *Linear control at another day than in* **Figure 3-7**

Of all different methods tried in this work the linear control was the one that took the most time to implement. This might seem a little bit strange but the problems were not the control problems but problems to get the different parts of the system working together, e.g. the Matlab compiler, the C-compiler or the data acquisition board. These problems will always appear in all kind of control. The difference was that when the adaptive control was made these functions already existed. Identification is difficult and such a thing as a perfect model does not exist. This leads to a steady state error but the integrator quite effectively compensates these nonlinearities.

3.3. Conclusions

From the different identifications it is easy to see that none of the models are valid at every time. Identification is more about getting a model as good as possible than getting the perfect one. The automatic control field is built on the idea that making the model as irrelevant as possible. Feedback is all about correcting the errors in a model. Even if the model is not exactly correct it is possible to obtain good control behaviour. The integrator is to good help in the work of reducing the steady-state error.

Even if the control is acceptable in some cases the model parameters have to be updated if the behaviour of the process is changed much. This was to bee seen in Figure 3-7 and Figure 3-8 when the control behaviour was completely different at two different days. The adaptive control will provide help for this kind of problems.

4. Adaptive control theory

4.1. Adaptive control schemes

4.1.1. Introduction to self tuning controllers

Sometimes you do not have an accurate model or the process will most likely change its behaviour. In these cases it is recommended to use an extended system that continuously update the polynomials $A(q^{-1})$ and $B(q^{-1})$. These kinds of systems are called adaptive systems.

There are many different approaches to update the model. In this report only updating with the use of the covariance matrix is used. The idea is to make an estimation of the process polynomials, with the help of the inputs and outputs of the process. This estimation is then used to calculate a controller from the given specifications. This is a quite straightforward method that is often used.

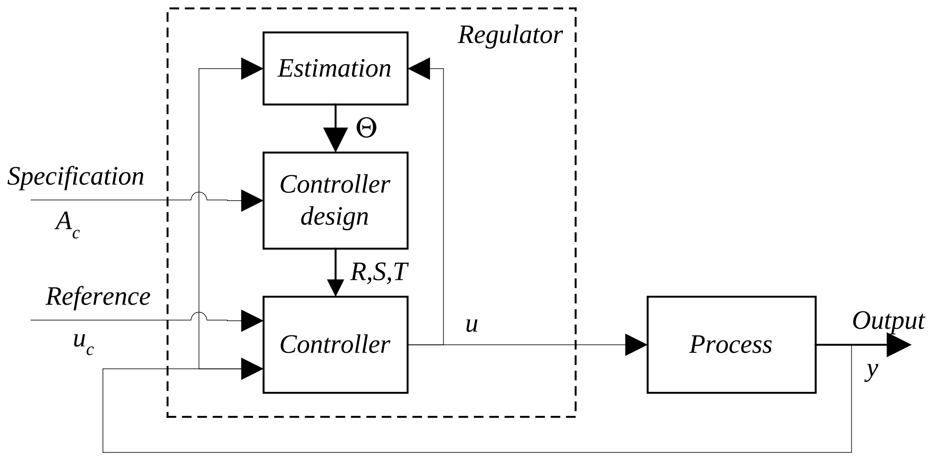


Figure 4-1 *A schematic picture of a adaptive control system.*

The estimation part of the controller estimates the process polynomials and the controller design uses these polynomials to put the closed loop poles according to the specifications. The controller is the same as in the linear case except for the constant regulator polynomials are changed to the ones calculated from the controller design.

Sometimes the control problem is slightly different from what was mentioned above. If the process does not change its behaviour, but the model parameters are unknown, a controller that estimates these is needed. This is a slightly easier problem because a fully adaptive controller is also able to adjust its parameters when the process changes its behaviour. The solutions are similar but the distinction has to be made. This type of controllers is called self-tuning.

4.1.2. Mathematical methods for adaptive control

The ideal way to estimate the polynomials is to adjust the model slightly at every sample. This is also possible and these kinds of methods are called recursive methods. One of the key elements in recursive methods is the parameter adaptation algorithm. This algorithm calculates the new model from the last model together with the last data. The structure is always the following (Landau, 1997):

$$\begin{bmatrix}\text{New parameters}\\ \text{estimation (vector)}\\ \Theta(k)\end{bmatrix}=\begin{bmatrix}\text{Previous parameters}\\ \text{estimation (vector)}\\ \Theta(k\text{-}1)\end{bmatrix}+\begin{bmatrix}\text{Adaptation gain}\\ \text{(matrix)}\\ F(k)\end{bmatrix}*\begin{bmatrix}\text{Measurement}\\ \text{function (vector)}\\ \phi^{\mathrm{T}}(k)\end{bmatrix}*\begin{bmatrix}\text{Prediction error}\\ \text{function (scalar)}\\ \varepsilon(k)\end{bmatrix}$$

The measurement function is also called the observation vector.

The most usual parameter adaptation algorithm is the gradient parameter adaptation algorithm. This algorithm minimizes the quadratic prediction error. Consider the simple system:

$$y(k+1)=-a_1y(k)+b_1u(k)=\Theta^T\phi(k)$$

where both a_1 and b_1 are unknown. This means that $\Theta^T=[a_1 \quad b_1]$ is the parameter vector and $\phi^T(k)=[-y(k) \quad u(k)]$ is the observation vector.

The priori prediction is:

$$\hat{y}^o(k+1)=\hat{\Theta}^T(k)\phi(k) \qquad \textbf{4-1}$$

where $\hat{\Theta}^T(k)$ is the predicted model at time k and the priori prediction error is

$$\varepsilon^o(k+1) = y(k+1) - \hat{y}^o(k+1) . \qquad \textbf{4-2}$$

The structure of the parameter adaptation algorithm is:

$$\hat{\Theta}(k+1) = \hat{\Theta}(k) + f\left[\hat{\Theta}(k), \phi(k), \varepsilon^o(k+1)\right] \qquad \textbf{4-3}$$

where the last term is called the correction term. One way to choose the correction term to is make it fulfil the following criterion:

$$\min_{\hat{\Theta}(k)} J(k+1) = \left[\varepsilon^o(k+1)\right]^2 . \qquad \textbf{4-4}$$

The fact that in a plane of the parameters a_1 and b_1 the minimum of the loss function is surrounded by concentric closed curves where *J*=constant. To find the minimum you move in the opposite direction of the gradient of the isocriterion curve (see Figure 4-2). This means:

$$\hat{\Theta}(k+1) = \hat{\Theta}(k) - F(k)\frac{\partial J(k+1)}{\partial \hat{\Theta}(k)} \qquad \textbf{4-5}$$

where *F* is the adaptation gain. The most used name for *F* is probably the covariance matrix.

From (eq. 4-4) one obtains:

$$\frac{1}{2}\frac{\partial J(k+1)}{\partial \hat{\Theta}(k)} = \frac{\partial \varepsilon^o(k+1)}{\partial \hat{\Theta}(k)}\varepsilon^o(k+1) .$$

Moreover is

$$\varepsilon^o(k+1) = y(k+1) - \hat{\Theta}(k)^T \phi(k)$$

and

$$\frac{\partial \varepsilon^o(k+1)}{\partial \hat{\Theta}(k)} = -\phi(k).$$

This means that

$$\hat{\Theta}(k+1) = \hat{\Theta}(k) + F\phi(k)\varepsilon^o(k+1). \qquad \textbf{4-6}$$

There are two ways to choose the covariance matrix.

1. $F=\alpha I;\ \alpha>0$ and I is the identity matrix.
2. F is a positive definite matrix. (All terms in main diagonal are positive, F is symmetric and the determinant of all the principal minors is positive)

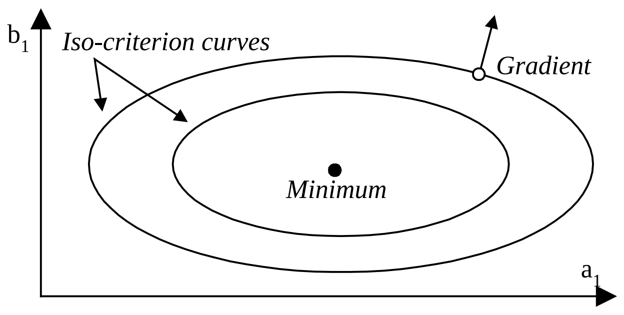


Figure 4-2 *The negative gradient of the iso-criterion curve is pointing in the general direction of the optimal estimation of the model.*

With a too large adaptation gain there is risk for instability. Another way to build the adaptation gain is to use the posterior prediction. The posterior prediction is:

$$\hat{y}(k+1) = \hat{\Theta}(k+1)^T \phi(k) \qquad \textbf{4-7}$$

and the posterior prediction error is

$$\varepsilon(k+1) = y(k+1) - \hat{y}(k+1). \qquad \textbf{4-8}$$

This leads to the loss function:

$$\min_{\hat{\Theta}(k+1)} J(k+1) = [\varepsilon(k+1)]^2 . \qquad \textbf{4-9}$$

The equations for the posterior case are then:

$$\frac{1}{2}\frac{\partial J(k+1)}{\partial \hat{\Theta}(k+1)} = \frac{\partial \varepsilon^o(k+1)}{\partial \hat{\Theta}(k+1)} \varepsilon(k+1)$$

$$\varepsilon(k+1) = y(k+1) - \hat{\Theta}(k+1)^T \phi(k)$$

$$\frac{\partial \varepsilon(k+1)}{\partial \hat{\Theta}(k+1)} = -\phi(k)$$

The final solution is then:

$$\hat{\Theta}(k+1) = \hat{\Theta}(k) + F\phi(k)\varepsilon(k+1) \qquad \textbf{4-10}$$

The problem is that $\varepsilon(k+1)$ is unknown. We need a function of $\varepsilon^o(k+1)$. Known is:

$$\varepsilon(k+1) = y(k+1) - \hat{\Theta}(k)^T \phi(k) - \left[\hat{\Theta}(k+1) - \hat{\Theta}(k)\right]^T \phi(k) =$$
$$= \varepsilon^o(k+1) - \phi(k)^T F\phi(k)\varepsilon(k+1) = \frac{\varepsilon^o(k+1)}{1+\phi(k)^T F\phi(k)}$$

Finally: $\hat{\Theta}(k+1) = \hat{\Theta}(k) + \dfrac{F\phi(k)\varepsilon^o(k+1)}{1+\phi(k)^T F\phi(k)}$ **4-11**

Just because the quadratic error is minimised at every sample it does not mean that the sum of the errors are minimised. The least square criterion is:

$$\min_{\hat{\Theta}(k)} J(k) = \sum_i \left[y(i) - \hat{\Theta}(k)^T \phi(i-1)\right]^2 \qquad \textbf{4-12}$$

This criterion is fulfilled when:

$$\frac{\partial J(k)}{\partial\hat{\Theta}(k)} = -2\sum_{i=1}^{k}\left[y(i) - \hat{\Theta}(k)^T\phi(i-1)\right]\phi(i-1) = 0$$

From:

$$\left[\hat{\Theta}^T(k)\phi(i-1)\right]\phi(i-1) = \phi(i-1)\phi(i-1)^T\hat{\Theta}(k)$$

one obtains:

$$\left[\sum_{i+1}^{k}\phi(i-1)\phi(i-1)^T\right]\hat{\Theta}(k) = \sum_{i=1}^{k}y(i)\phi(i-1)$$

and by left multiplying with:

$$\left[\sum_{i=1}^{k}\phi(i-1)\phi(i-1)^T\right]^{-1}$$

one obtains:

$$\hat{\Theta}(k) = \left[\sum_{i-1}^{k}\phi(i-1)\phi(i-1)^T\right]^{-1}\sum_{i=1}^{k}y(i)\phi(i-1) = F(k)\sum_{i=1}^{k}y(i)\phi(i-1)$$

in which:

$$F(k)^{-1} = \sum_{i=1}^{k}\phi(i-1)\phi(i-1)^T$$

$\hat{\Theta}(k+1)$ is needed to get an recursive algorithm:

$$\hat{\Theta}(k+1) = F(k+1)\sum_{i=1}^{k+1}y(i)\phi(i-1)$$

$$F(k+1)^{-1} = \sum_{i=1}^{k+1}\phi(i-1)\phi(i-1)^T = F(k)^{-1} + \phi(k)\phi(k)^T$$

$\hat{\Theta}(k+1)$ is expressed as a function of $\hat{\Theta}(k)$ on the form:

$$\hat{\Theta}(k+1) = \hat{\Theta}(k) + \Delta\hat{\Theta}(k+1)$$

$$\sum_{i=1}^{k+1} y(i)\phi(i-1) = \sum_{i=1}^{k} y(i)\phi(i-1) + y(k+1)\phi(k)$$

This can be written as:

$$\begin{aligned}&\sum_{i=1}^{k+1} y(i)\phi(i-1) = F(k+1)^{-1}\hat{\Theta}(k+1) = \\ &= F(k)^{-1}\hat{\Theta}(k) + \phi(k)\phi(k)^T\hat{\Theta}(k) + \phi(k)\left[y(k+1) - \hat{\Theta}(k)^T\phi(k)\right]\end{aligned}$$

Further on:

$$F(k+1)^{-1}\hat{\Theta}(k+1) = F(k+1)^{-1}\hat{\Theta}(k) + \phi(k)\varepsilon^o(k+1)$$

After a multiplication from the left by: *F(t+1)* the result is:

$$\hat{\Theta}(k+1) = \hat{\Theta}(k) + F(k+1)\phi(k)\varepsilon^o(k+1) \qquad \textbf{4-13}$$

F(k+1) is given by the matrix inversion lemma (Landau, 1990):

$$\left(F^{-1} + \phi\phi^T\right)^{-1} = F - \frac{F\phi\phi^T F}{1 + \phi(k)^T F(k)\phi(k)}$$

From this lemma one obtains:

$$F(k+1) = F(k) - \frac{F(k)\phi(k)\phi(k)^T F(k)}{1 + \phi(k)^T F(k)\phi(k)} \qquad \textbf{4-14}$$

An equivalent description can be obtained as:

$$\left[\hat{\Theta}(k+1)-\hat{\Theta}(k)\right]=F(k+1)\phi(k)\varepsilon^{o}(k+1)=F(k)\phi(k)\frac{\varepsilon^{o}(k+1)}{1+\phi(k)^{T}F(k)\phi(k)}$$

However:

$$\begin{aligned}&\varepsilon(t+1)=y(t+1)-\hat{\Theta}(t+1)^{T}\phi(t)=y(t+1)-\hat{\Theta}(t)\phi(t)-\\&\left[\hat{\Theta}(t+1)-\hat{\Theta}(t)\right]^{T}\phi(t)=\varepsilon^{o}(k+1)-\phi(k)^{T}F(k)\phi(k)\frac{\varepsilon^{o}(k+1)}{1+\phi(k)^{T}F(k)\phi(k)}=\\&=\frac{\varepsilon^{o}(k+1)}{1+\phi(k)^{T}F(k)\phi(k)}\end{aligned} \qquad \textbf{4-15}$$

which is an expression with the posterior estimation.

To sum up the most important equations:

$$\hat{\Theta}(k+1)=\hat{\Theta}(k)+F(k)\phi(k)\varepsilon(k+1)$$

$$F(k+1)=F(k)-\frac{F(k)\phi(k)\phi(k)^{T}F(k)}{1+\phi(k)^{T}F(k)\phi(k)}$$

$$\varepsilon(k+1)=\frac{y(k+1)-\hat{\Theta}(k)^{T}\phi(k)}{1+\phi(k)^{T}F(k)\phi(k)}$$

As a initial value of the covariance matrix,

$$F(0)=\frac{1}{\delta}I,\quad 0<\delta<<1$$

is normally a good choice.

These equations are decreasing the adaptation gain in time. This can be seen if the estimation is only made on one parameter.

$$F(k+1)=\frac{F(k)}{1+\phi(k)^{2}F(k)}\leq F(k)$$

This means that less and less weight is given to new prediction errors. When the systems are varying in time the first, and inaccurate, predictions are the most important for the final estimation. A new estimation profile is therefore needed. We introduce the inverse of the adaptation gain with two forgetting profiles, $\lambda_1(k)$ and $\lambda_2(k)$:

$$F(k+1)=\frac{1}{\lambda_1(k)}\left[F(k)-\frac{F(k)\phi(k)\phi(k)^T F(k)}{\frac{\lambda_1(k)}{\lambda_2(k)}+\phi(k)^T F(k)\phi(k)}\right] \qquad \textbf{4-16}$$

The choice of the forgetting profiles can make the system behave in slightly different ways. To illustrate this some examples were given by (Landau, 1997):

<u>Decreasing gain (RLS)</u>

$$\lambda_1(k)=\lambda_2(k)=1 \qquad \textbf{4-17}$$

This is the forgetting profile without forgetting factors above, the least-mean-square. This profile is suitable for identification of stationary systems.

<u>Constant forgetting factor</u>

When working with constant forgetting factors λ_1 is usually the only active forgetting factor.

In this case:

$$\lambda_1(k)=\lambda_1; \quad 0<\lambda_1<1; \quad \lambda_2(k)=\lambda_2=1 \qquad \textbf{4-18}$$

Typical values for λ_1 are λ_1=0.95 to 0.99. The criterion to be minimized is:

$$J(k)=\sum_{i=1}^{k}\lambda_1^{(k-1)}\left[y(i)-\hat{\Theta}(k)^T\phi(i-1)\right]^2$$

The weight of the old data is getting lower and lower as time evolves. The highest weight is given to the most recent error. This kind of forgetting profile might be useful on slow varying systems.

Variable forgetting factor

There are many ways to choose variable forgetting factors. One is:

$$\lambda_2(k)=1; \quad \lambda_1(k)=\lambda_0\lambda_1(k-1)+1-\lambda_0; \quad 0<\lambda_0<1 \qquad \textbf{4-19}$$

Typical values are: $\lambda_1(0)$=0.95 to 0.99 and $\lambda_0(0)$=0.95 to 0.99.

The forgetting factor is asymptotically tending toward 1. The loss function will be:

$$J(k)=\sum_{i=1}^{k}\lambda_1(i)^{(k-i)}\left[y(i)-\hat{\Theta}(k)^T\phi(i-1)\right]^2$$

The forgetting factor will be one for after a while. This means that only the initial estimations are forgotten. For stationary systems this profile is to be preferred because too rapid decreasing of the adaptation gain is avoided. This is good because a high adaptation gain initially is good when the estimation is far from the optimum. The convergence is faster this way.

The trace of the covariance matrix is the sum of the elements in the main diagonal of the matrix and this can be used as a measure of the adaptation gain. In all the cases above using the forgetting factors the elements in the covariance matrix are increasing when the error is small. When the excitation is small, for example when the process has reached a steady state, the trace is going to increase until the system fails. This phenomenon is called estimator wind-up. To prevent this type of errors there are different methods that can be used.

Constant gain

The simplest way to track of the covariance matrix is to keep it constant.

$$F(k+1)=F(k)=F(0) \qquad \textbf{4-20}$$

This type of covariance profile is very simple to implement but is not very flexible. The only way to control the adaptation is to choose an initial value. In this method the covariance matrix does not store any information of the old measurements.

Constant trace

One way of controlling the covariance trace is to keep it constant.

$$trF(k+1) = trF(k) = trF(0) = nGI \qquad where\, F(0) = \begin{bmatrix} G & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & G \end{bmatrix} \tag{4-21}$$

Normal values for G=0.1 to 4.

The loss function is:

$$J(k) = \sum_{i=1}^{k} f(k,i)\left[y(i) - \hat{\Theta}(k)^T \phi(i-1)\right]^2$$

where *f(k,i)* represents the forgetting profile.

However the relation between *F(k+1)* and *F(k)* is calculated from:

$$trF(k+1) = \frac{1}{\lambda_1(k)} tr\left[F(k) - \frac{F(k)\phi(k)^T F(k)}{\dfrac{\lambda_1(k)}{\lambda_2(k)} + \phi(k)^T F(k)\phi(k)}\right] \tag{4-22}$$

To calculate the forgetting factors a relation between λ_1 and λ_2 is needed. One approach is to keep the quota λ_1/λ_2 constant and then calculate the λ_1.from the eq. 4-22.

This type of forgetting profile is useful for identification of time varying parameters. As long as the error is small the forgetting factors are small and when the model is changing the regulator forgets the old inaccurate model by decreasing the forgetting factors.

5. Experimental results with the adaptive control

5.1. Introduction

As always when it comes to identification it is very important that the identification that is made is an identification of the behaviour of the process and not the noise. The PT326 process changes its behaviour from day to day and even from hour to hour. These changes are a result of differences in the temperature of the surrounding air. There are also differences in how long time the process has been running. At one point the thermistor broke. After a new thermistor had been installed the parameters were changed drastically. This example is one reason why it is important to have a flexible controller that can deal with these kind of changes.

5.2. Simulation results

The simulations are made with badly chosen initial values and at sample 150 the parameters are changed. From this it is possible to see how the different covariance algorithms are working. All simulations were made without integrator. The initial model was $A(q^{-1}) = 1 - 1.5q^{-1} + 0.5q^{-2}$ and $B(q^{-1}) = 0.2q^{-1} + 0.5q^{-2}$. The output was before the load calculated with the model from the linear control: $A(q^{-1}) = 1 - 1.1628q^{-1} + 0.3095q^{-2}$ and $B(q^{-1}) = 0.1249q^{-1} + 0.1264q^{-2}$. After the parameters were changed they were calculated with the polynomials: $A(q^{-1}) = 1 - 1.5q^{-1} + 0.7q^{-2}$ and $B(q^{-1}) = 0.1q^{-1} + 0.4q^{-2}$.

5.2.1. Decreasing gain

The forgetting factors are all the time equal to one. This means that no information is forgotten and that all the information is weighted all the same. After the change of parameters the old data is never discarded. The result is a bad controller that always will have a steady state error. It could be used as a self-tuning regulator though. When the system is constant the controller do not have to forget any data because the model is the same all the time. The old errors are accurate all the time. When the disturbance occurs the regulator never completely forgets the old data. That leads to a steady-state error.

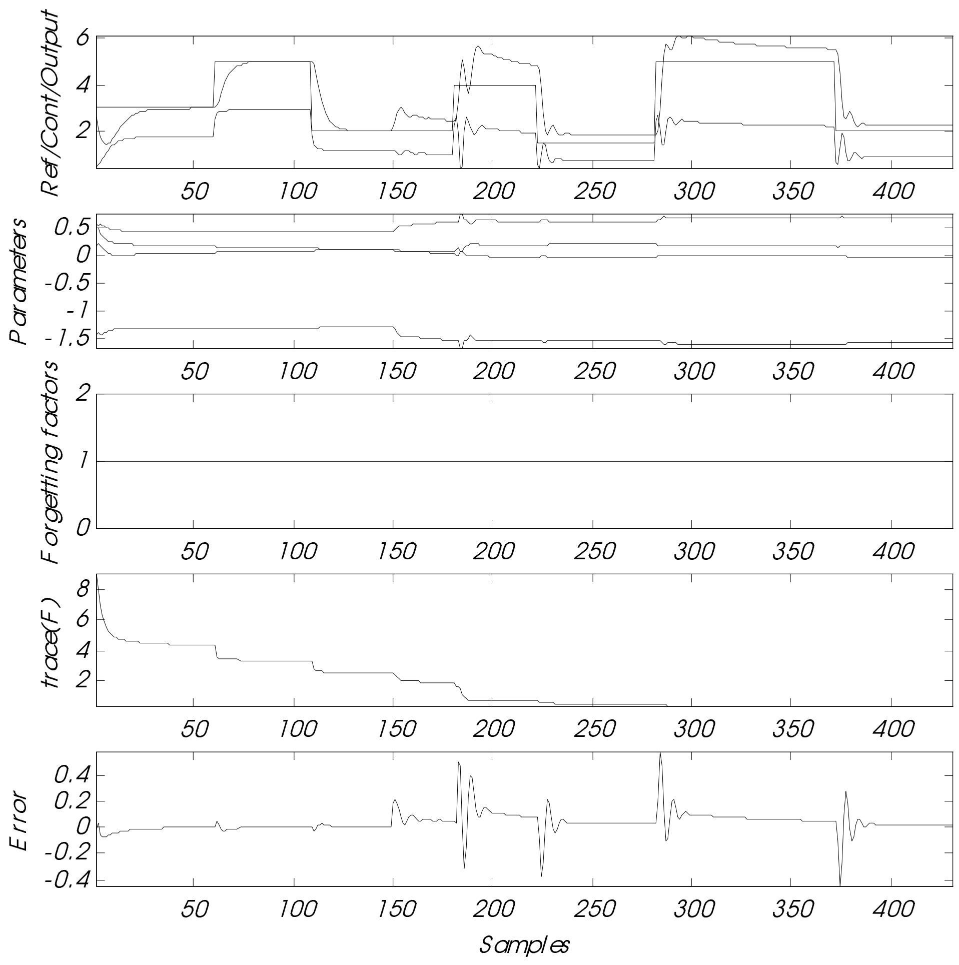


Figure 5-1 *Simulation results using decreasing gain. The forgetting factors are constantly at 1. All the data is used for the identification.*

5.2.2. Constant forgetting factor

With the constant forgetting factor $\lambda_1 = 0.95$ the control might look good. The signals look good and a correct model is developing quite fast. The steady state error goes towards zero. What is not seen is that at the end is the trace of the covariance matrix about 2000. This is to be compared with the initial 12. In a longer perspective this controller might break down.

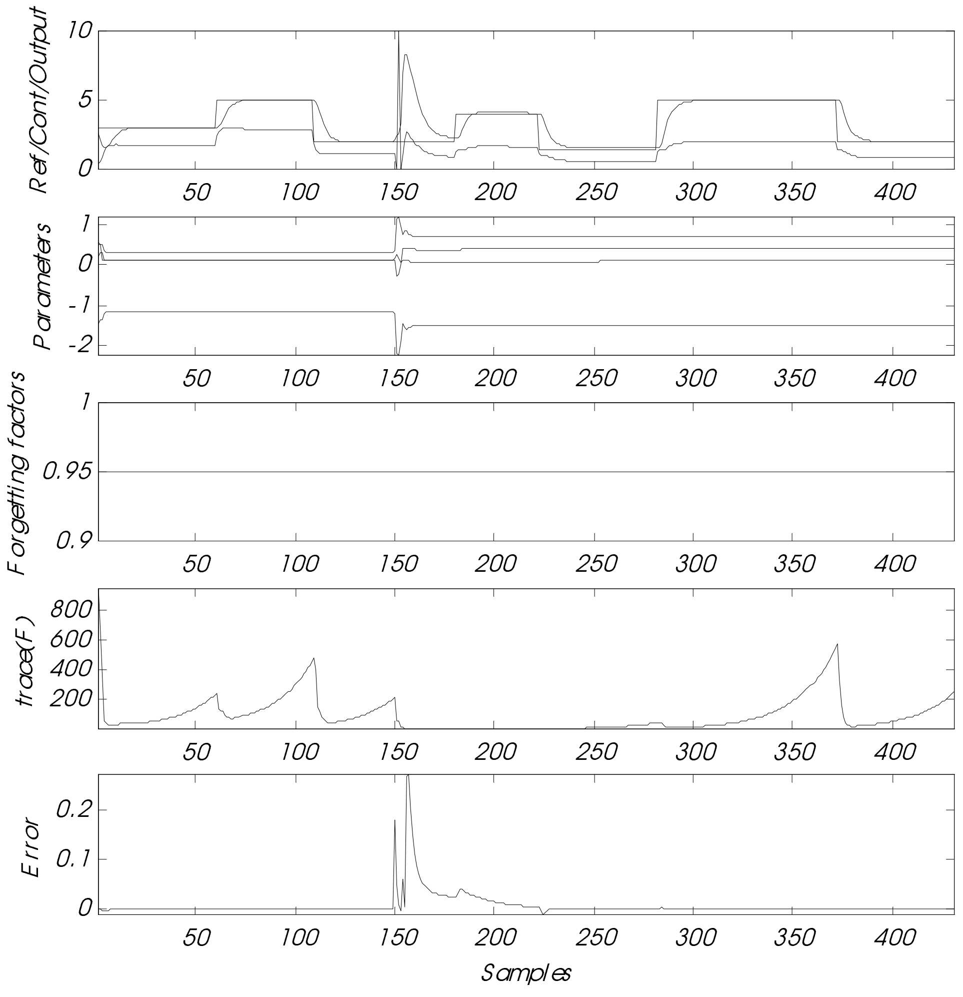


Figure 5-2 *Simulation results using constant forgetting factor. The old data is forgotten always at the same speed.*

5.2.3. Variable forgetting factor

The attempt to change the forgetting factor in the beginning is to no use if the process parameters are changed. The forgetting factor is a function of time and will quite rapidly go towards one. This means that when the variations occur the system do not forget anything. As a self-tuning regulator it would be perfectly all right due to the fact that the only things this type of regulator forgets is the initial bad values.

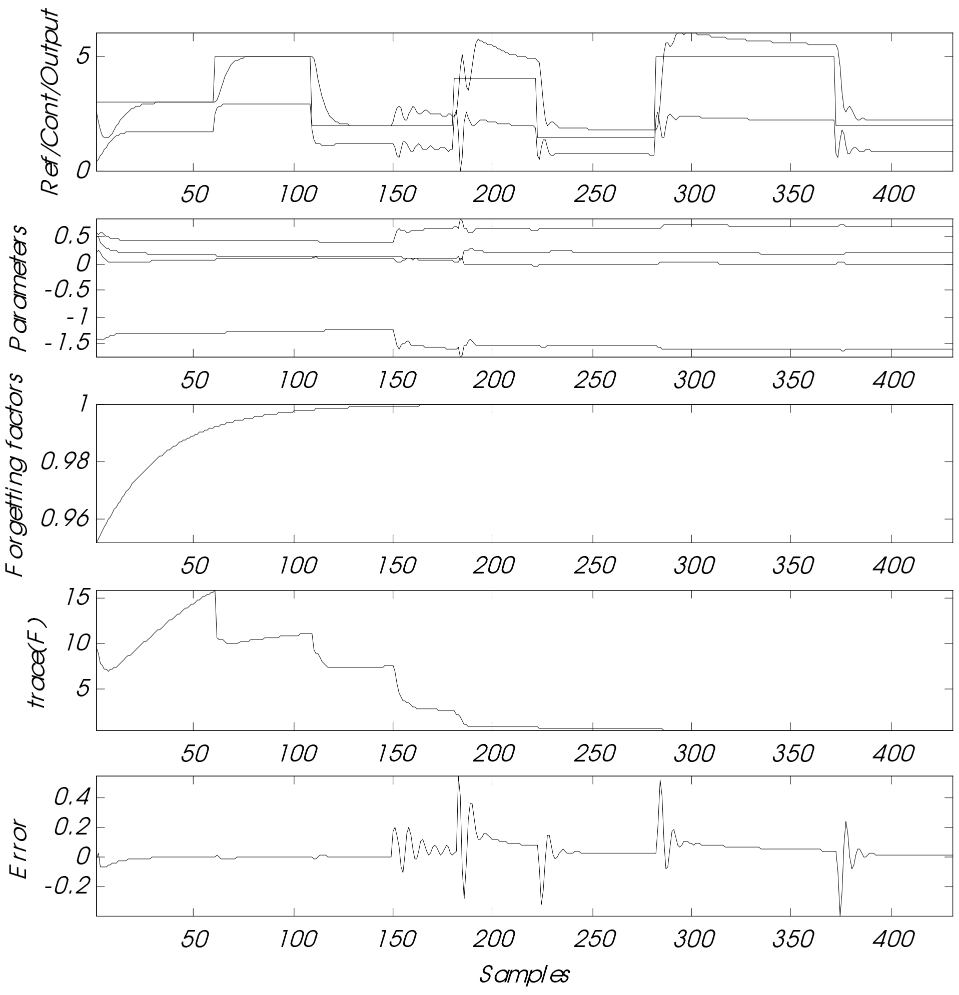


Figure 5-3 *Simulation results using the variable forgetting factor. Only the initial data is forgotten.*

5.2.4. Constant gain

As a first attempt to avoid wind up is by using the idea of keeping the covariance matrix constant. It is a very basic idea but in the simulations it seems to work even if the covariance matrix does not contain any information about measurements from the past.

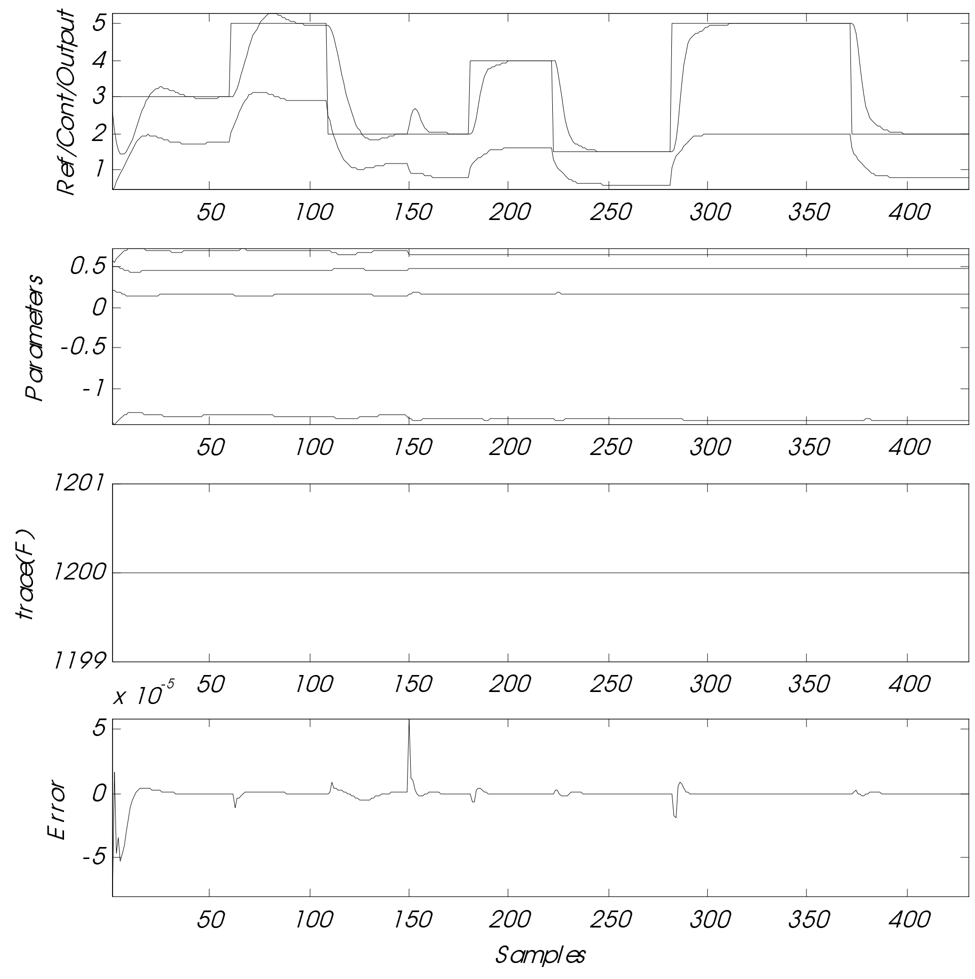


Figure 5-4 *Simulation results using constant gain. The covariance matrix is here a constant diagonal matrix.*

5.2.5. Constant trace

Simulations were made in two different ways. One where λ_2 was held constant equal to 1 and one where the quote, $\alpha=\lambda_1/\lambda_2$ was held constant. The result was more or less equal, why only the simulation with constant λ_2 is presented here.

Every time when an error occurs the regulator thinks that the model is not accurate, and therefore decrease the forgetting factor to create a new better model. This is working very well. At the end there is no steady state error and the adaptation is running fast. This is the type of regulator that showed the best simulations of all the others.

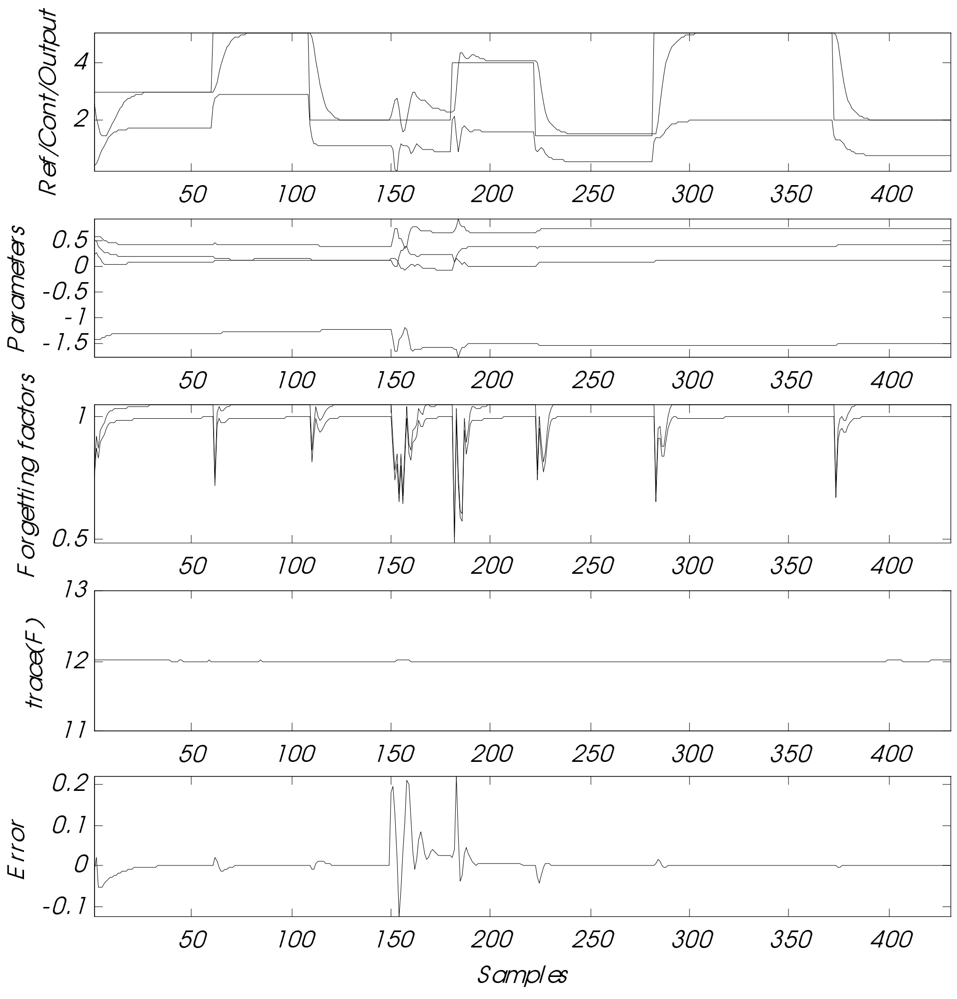


Figure 5-5 *Simulation results using the constant trace algorithm. The data is more rapidly forgotten when an error is detected in the model.*

5.3. Experimental results

5.3.1. General explanation to the experiments

The experiments were made without integrator to get the best picture of how good the model is. With an integrator the steady-state error is zero and compensates a bad model a lot. At the end an experiment was made with the constant trace algorithm and an integrator to see the difference.

At an approximate time, *k=120*, the entrance for the air is almost completely closed. The different controllers react differently to this change.

The trace has to be much lower in the practical case than in the simulations. If not the noise will be very influential in the parameter estimation.

Before the change the output signal is below the control signal and after the change the two switch places. The output is then situated above the control signal.

5.3.2. Decreasing gain

The result of the result can bee seen in Figure 5-6. In the beginning the controller tries to compensate its bad parameters. It is not very successful. When the changes in the process parameters occur, at the time *k=120*, the model is almost fixed to its values. The lack of change can be seen at the trace of the covariance matrix that is almost 0.

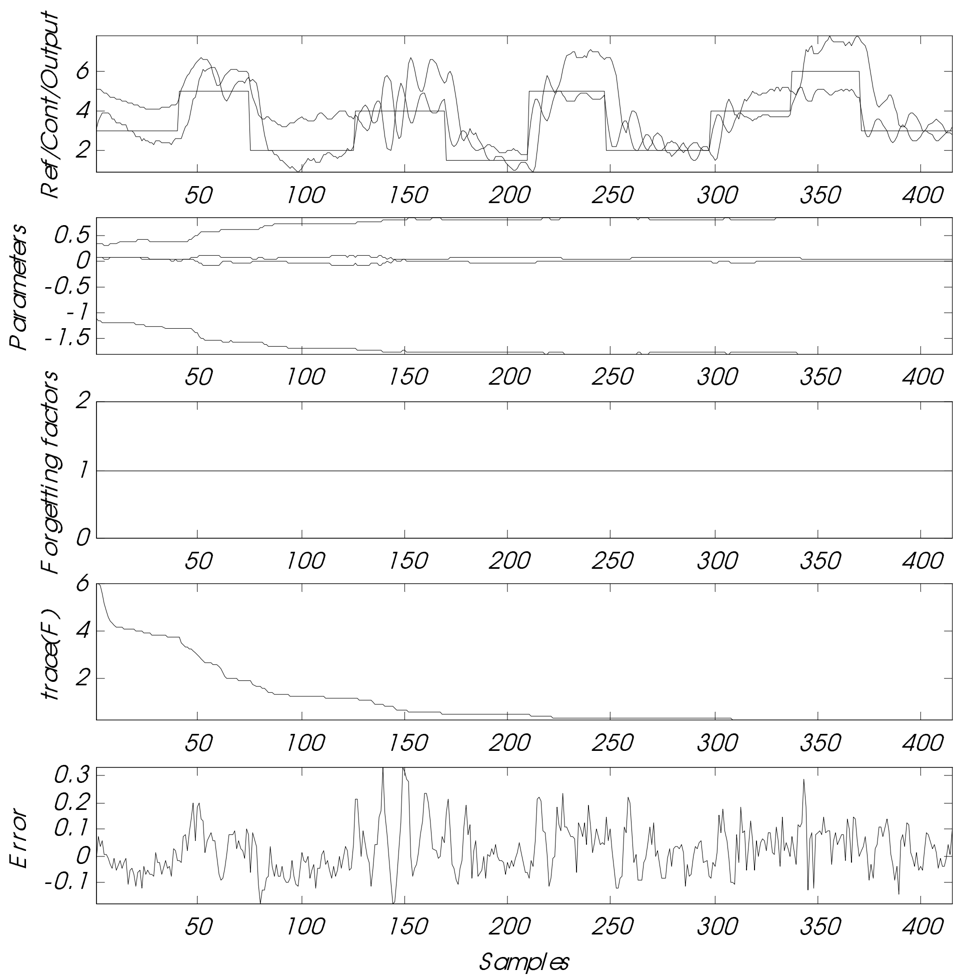


Figure 5-6 ***The decreasing gain controller.*** *In the top graph the output is the line that at the beginning is beneath the control signal. The decreasing gain controller does not forget any old measurements. When changes occur in the process the controller tends to keep the old model.*

5.3.3. Constant forgetting factor

In Figure 5-7 the result from the experiment with the controller using constant forgetting factors can be seen. At the beginning this controller behaves more or less as the decreasing gain controller. When the hatch is closed for the air, at time *k=110*, the trace has become low. Therefore the parameters are changed only slightly, even if the error is big. They are changing enough to decrease the error, which is making the trace increase. At the end the output is following the reference quite well. A new change in the process would be rapidly corrected, as long the as noise would make the elements in the covariance matrix too small, as when the inlet was decreased in this experiment. The modification of the parameters is very slow and the noise is affecting the result in a non-neglect able way.

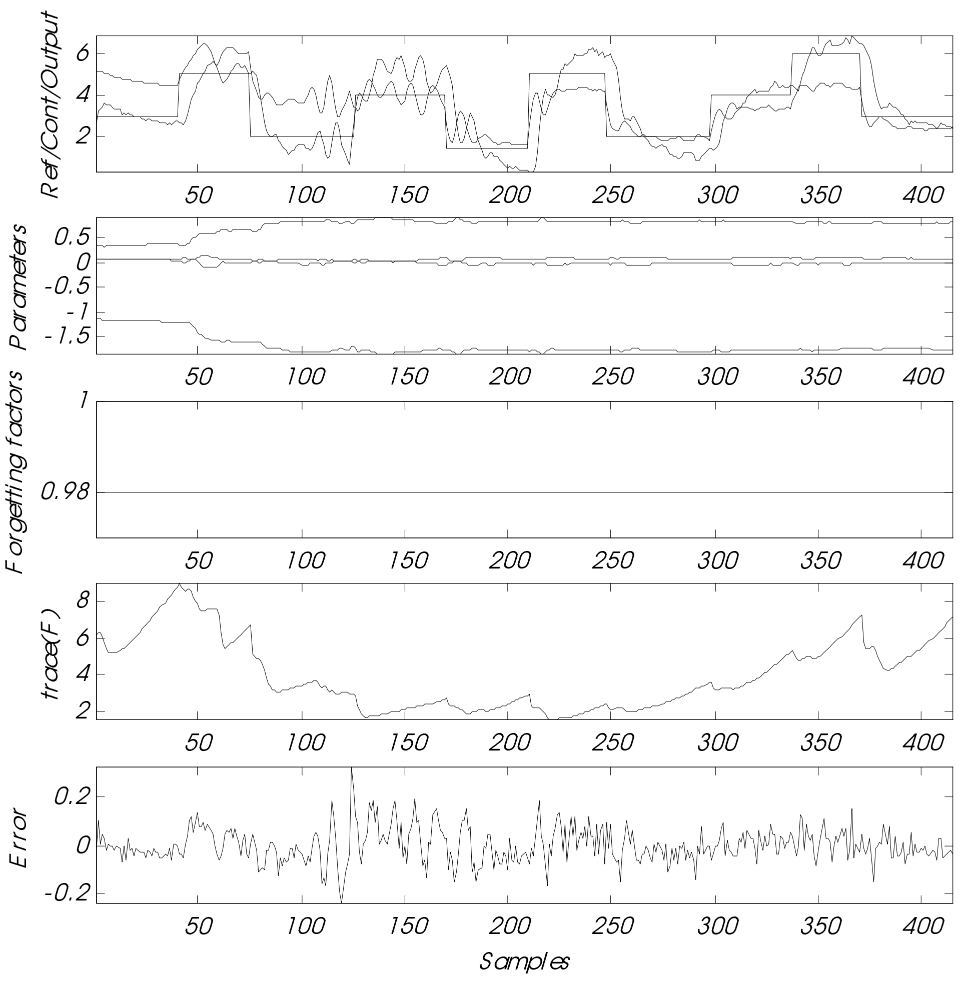


Figure 5-7 *The controller with constant forgetting factors. At the beginning the output of the process is the closest to the reference. At the end the output is almost following the reference.*

5.3.4. Variable forgetting factor

With the controller using variable forgetting factors the result was as to be seen in Figure 5-8. The forgetting factors are rapidly changing towards 1. Automatically the trace is getting close to 0. This means that when the variations in process parameters occur, at *k=120,* the model is not changing. This leads to a bad controller.

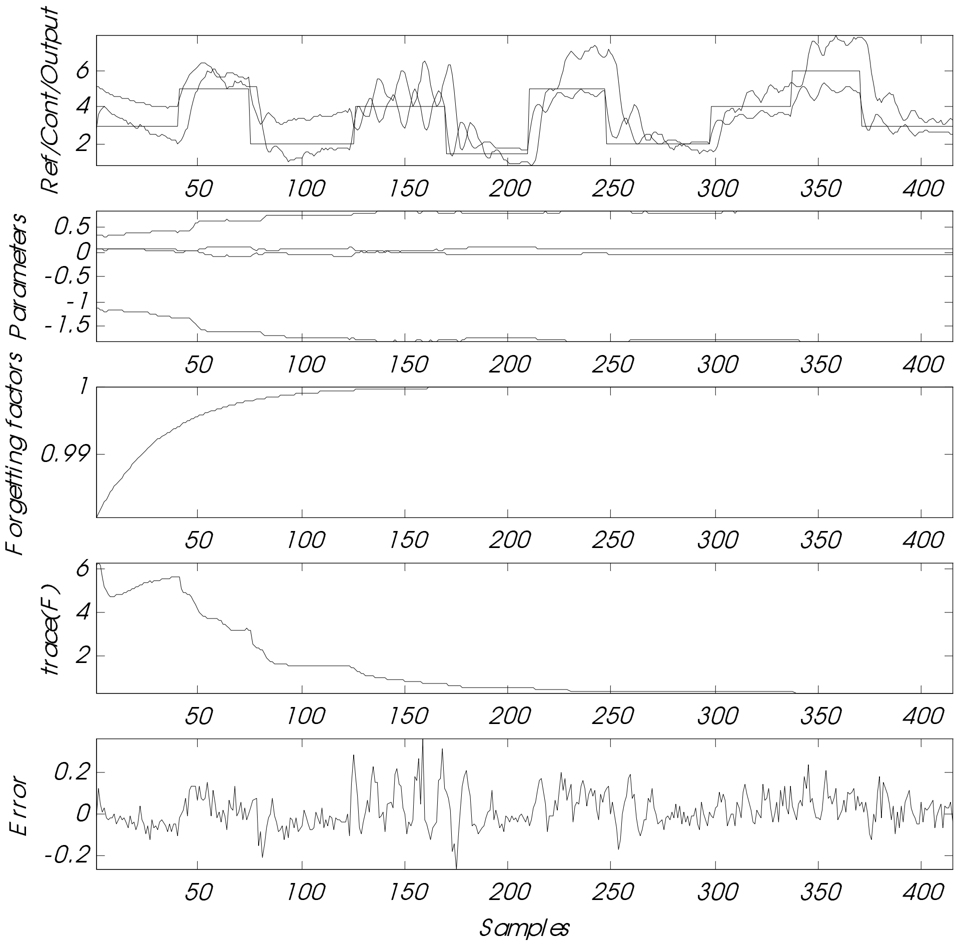


Figure 5-8 *The controller using variable forgetting factor. When the changes occur, the controller does not change the model any more.*

5.3.5. Constant gain

Surprisingly this type of controller showed the best results of all, as to be seen in Figure 5-9. The errors are rapidly decreased, after the closing of the hatch, to small values and the output is following the reference very well. The reason why the constant gain regulator was the best one is very difficult to explain. One suggestion is that the adaptation gain is constant. If it is well chosen the convergence is quick without getting unstable. All the other methods are limited because of the risk that the gain is being to big. If this gain is limited in some way the adaptation might be more efficient. More studies are needed before a conclusion is made. The limitation might be that none of the elements in the covariance matrix are allowed to exceed a certain value.

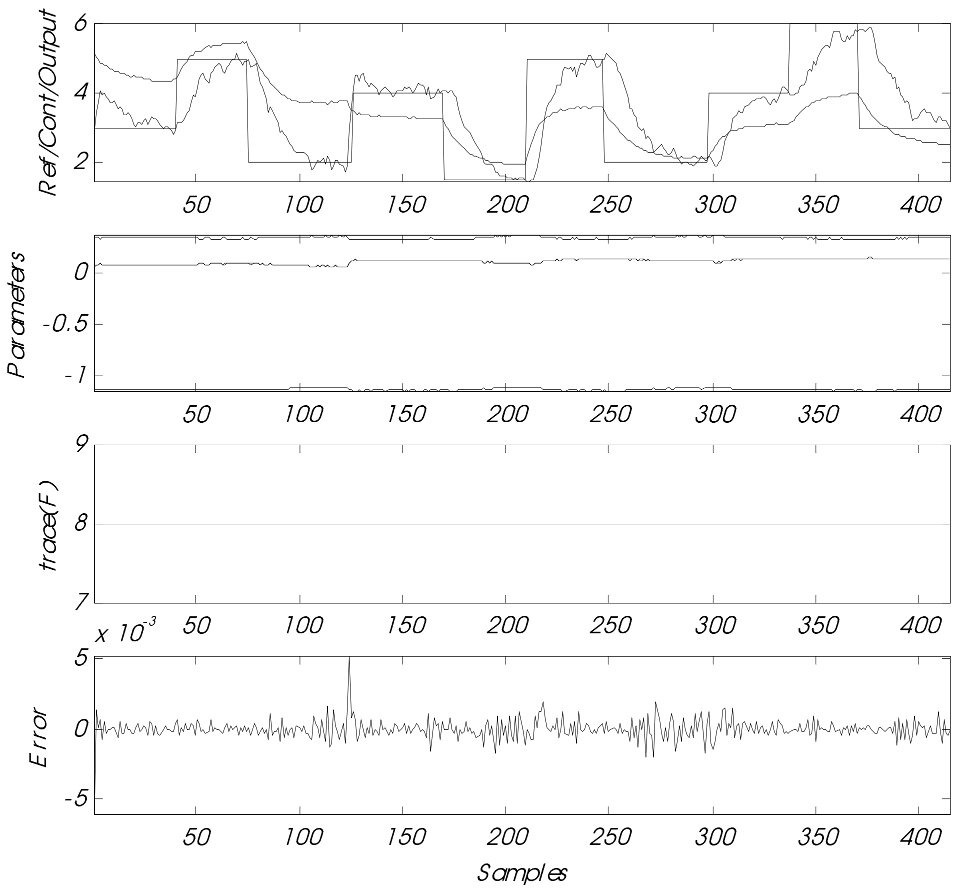


Figure 5-9 *The constant gain controller. This was the best controller of all the adaptive ones.*

5.3.6. Constant trace

The result of the experiments with the constant trace regulator can be seen in Figure 5-10. When the changes in process parameters occur, at time *k=120*, the controller changes the parameters rapidly. The problem is that if the changes have to be fast there is a risk that the model will be badly chosen, due to the low forgetting factors. As a matter of fact the noise is here enough to keep the forgetting factors at a low level, so the parameters are calculated from only few measurements. In an environment this would have been working better. Even so, here is the output almost following the reference without steady state errors.

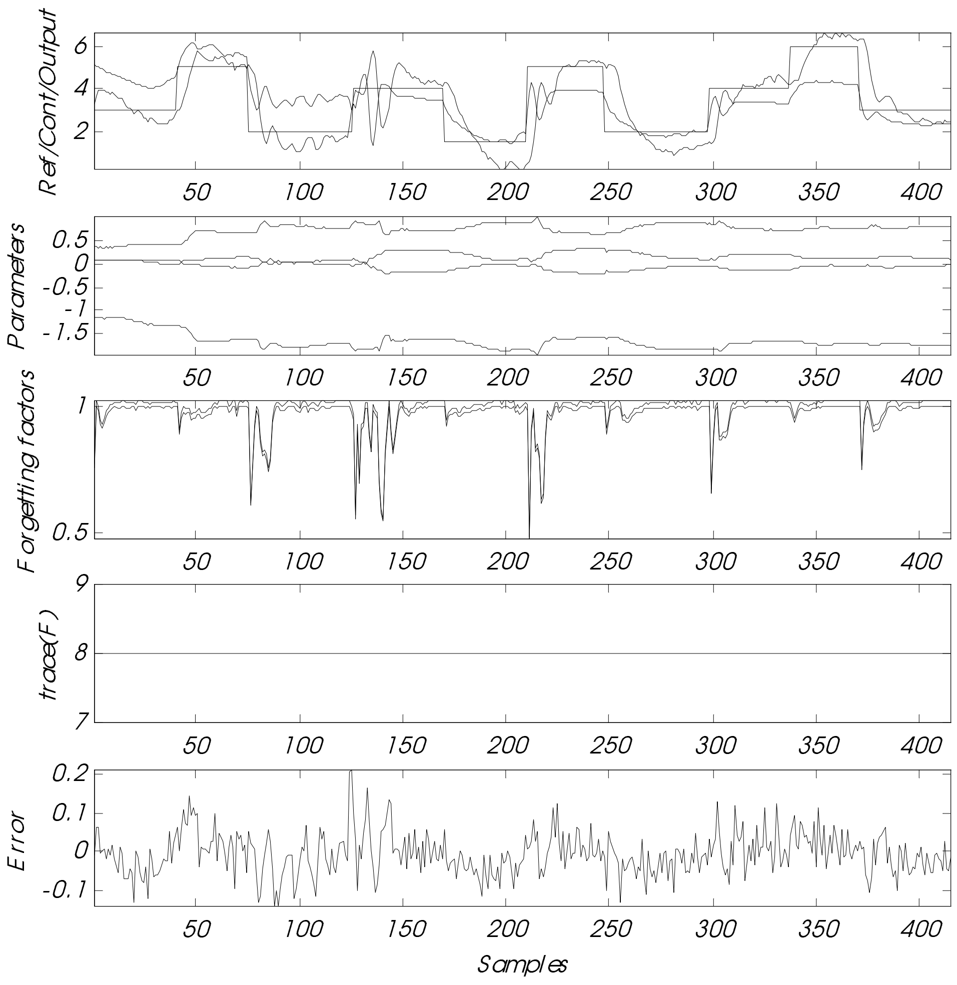


Figure 5-10 *The constant trace controller. With low forgetting factors when the error is big, the parameters are rapidly modified.*

To see how this kind of controller is working together with a integrator one more experiment was made. The result can be seen in Figure 5-11. This is how a constant trace regulator would look like if it were to be implemented in a more industrial environment. The result is more or less as good as it gets with a noisy process. Maybe one idea would be to decrease the trace a little bit so the forgetting factors did not go as low as they do. The adaptation would then be slightly slower.

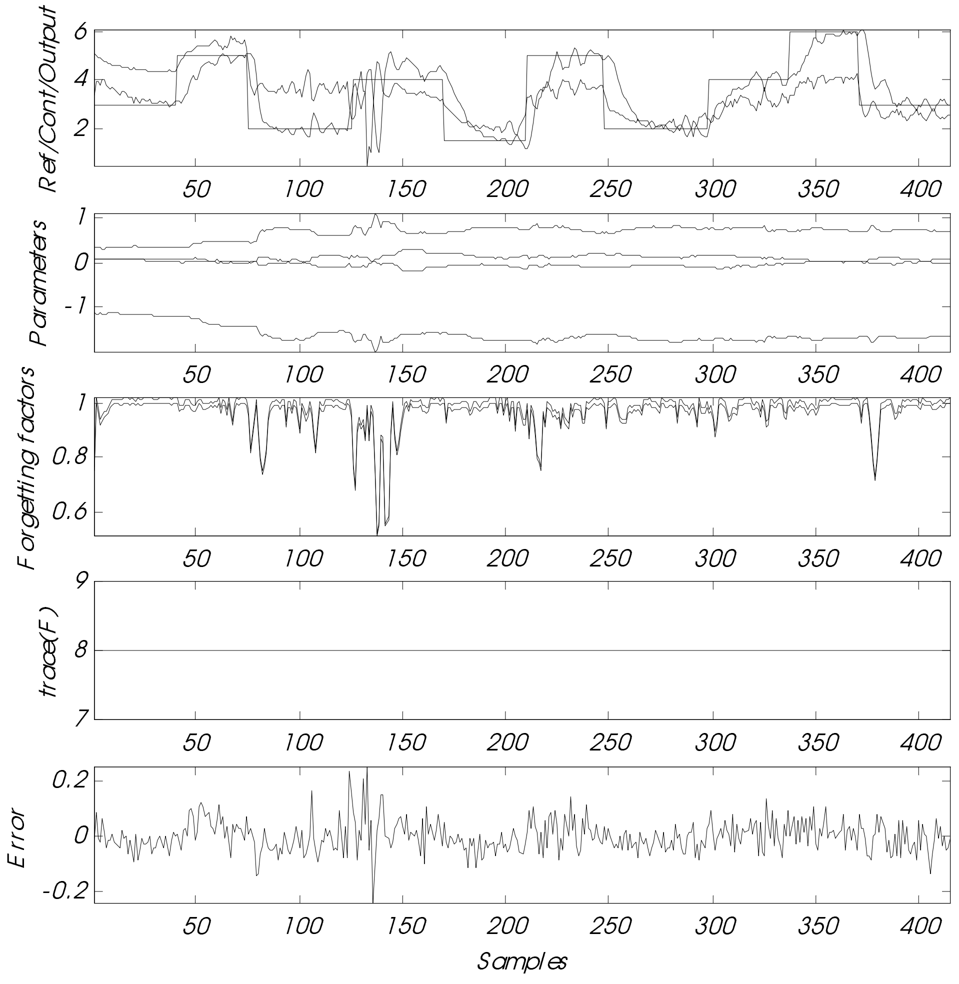


Figure 5-11 *The constant trace controller with an integrator.*

6. Neural networks

6.1. Structure of the text

The neural network technologies are a very wide area in engineering. There are several different types of nets and just a few will be covered here.

To illustrate the functions of a neural network the first type of nets are the Hopfield neural networks that slowly will be transformed into feed forward networks and then to recurrent neural networks.

6.2. Some basic neural network thinking by explaining Hopfield networks

6.2.1. Neuron and states

To explain the functions of a neural network the Hopfield networks will be explained.

The neural networks can be seen as multiple processors working in parallel. These processors are called neurons. None of the neurons contain more information than the others. Further on there is no central neuron that controls the others. The knowledge is distributed over all the neurons.

Today the biggest neural nets include a few hundred neurons.

A step-to-step approach will be used here to show the principals of a neural net.

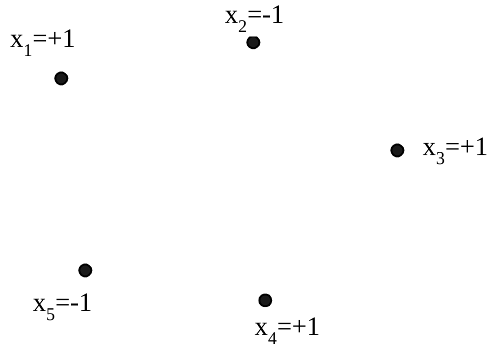


Figure 6-1 *Some neurons with their states*

In Figure 6-1 the neurons are represented by dots. To each neuron there is a number attached. This number can be either –1 or 1. Some people prefer 0 and 1, but in reality it makes no difference. This number is called the state of the neuron. For example: The state of neuron 4 is denoted x_4 and $x_4 = 1$. The states of all neurons can be collected in the state vector $\mathbf{x} = (x_1, x_2, x_3, x_4, x_5)^T$.

The neural networks are dynamical. This means that the states change in time. To illustrate the state vector the following notation is used at the time k: $\mathbf{x}$(k). $\mathbf{x}$(0) indicates the initial value.

Because the state vector has as many components as there are neurons, n, it is possible to define the state space as the n-dimensional space that is created from the possible values of the state vector. The neurons can as known be –1 or +1, in Hopfield neural networks. This means that the state space is the corners of an n-dimensional hypercube. This is where the similarities with control problems begin. When dealing with neural net we are using a state space model as a base. More of this is to be read further down.

6.2.2. Updating the neurons

The neurons influence each other according to an update rule. In the example the upgrade rule for x_4 is:

$$x_4(k+1) = \mathbf{sgn}(w_{41}x_1(k) + w_{42}x_2(k) + w_{43}x_3(k))$$

w is a weight on how important a certain neuron is for updating of another. The weights can be any real value. Sometimes the weights are called synapses. A more general way to describe the upgrade rule is:

$$x_i(k+1) = \mathbf{sgn}\left(\sum_{j=1}^{n} w_{ij}x_j(k)\right) \tag{6-1}$$

where n is the number of the neurons.

W is a matrix with the following form:

$$W = \begin{pmatrix} 0 & w_{12} & w_{13} & \cdots & w_{1,n} \\ w_{21} & 0 & w_{23} & & \vdots \\ w_{31} & w_{32} & 0 & \ddots & \vdots \\ \vdots & & \ddots & 0 & w_{n-1,n} \\ w_{n,1} & \cdots & \cdots & w_{n,n-1} & 0 \end{pmatrix}$$

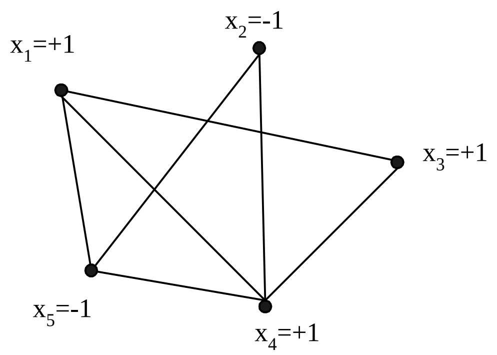


Figure 6-2 *Connections between the neurons in* **Figure 6-1**.

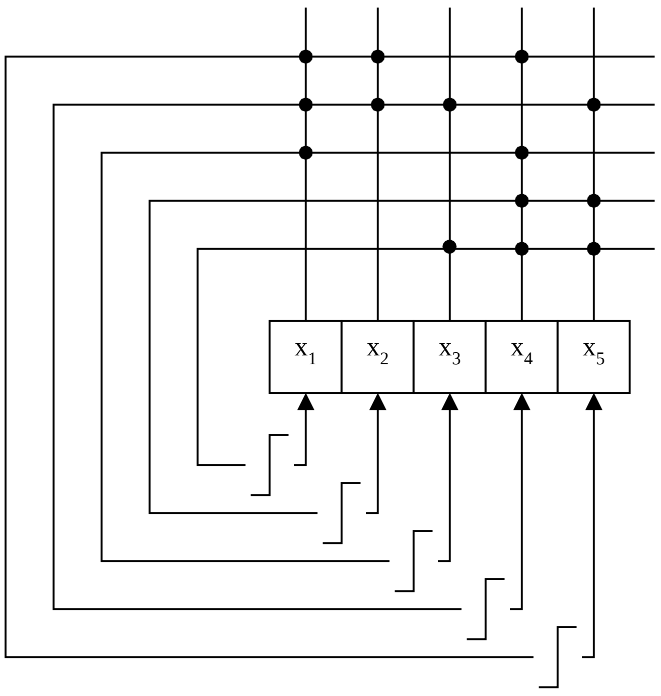


Figure 6-3 *The same connections as in* **Figure 6-2**, *but with a different notation.*

The weight matrix for a Hopfield network is symmetric and has zero diagonal. This means that the neuron x_1 influence x_2 in the same way as x_2 influence x_1 and no neuron influence itself. The reason for this will be mentioned in 6.2.3. This assumption will be changed for the other networks.

If all the neurons are updated at the same time the update is called synchronous and if the neurons are updated one at the time it is called asynchronous.

6.2.3. Associative memories

One way to describe the Hopfield networks nets are as associative memories. This means that if you for example store a number of photos in a memory you can show a picture of a person to the net and the net identifies the person and gives back some data about him. If the weights are well chosen, the initial picture might be filtered too. How this works will be explained in further down.

Assume the following evolution in time:

$$\begin{gathered}\mathbf{x}(0)=(+1-1+1+1-1)^T\\ \downarrow\\ \mathbf{x}(1)=(+1+1+1+1-1)^T\\ \downarrow\\ \mathbf{x}(2)=(+1+1+1-1-1)^T\\ \downarrow\\ \mathbf{x}(3)=(+1+1+1-1-1)^T\\ \downarrow\end{gathered}$$

After *k=2* is the state vector constant. The network has converged. All networks do not converge. Another example:

$$\begin{gathered}\mathbf{x}(0)=(+1+1+1-1-1)^T\\ \downarrow\\ \mathbf{x}(1)=(-1-1-1-1+1)^T\\ \downarrow\\ \mathbf{x}(2)=(+1+1+1-1-1)^T\\ \downarrow\\ \mathbf{x}(3)=(-1-1-1-1+1)^T\\ \downarrow\end{gathered}$$

In this example the state vector is oscillating between two states. To avoid this the weight matrix *W* in the Hopfield networks have zero diagonal and is symmetric.

The sequence of states as time evolves is called trajectory.

The endpoint of a trajectory is called fundamental memories or attractors. All states that are leading to the same fundamental memory are said to be in the same attraction basin. See Figure 6-4.

A way to decide the capacity of a neural network is by the number of fundamental memories is possible to store in it. This capacity is more or less linear to the number of neurons in the net. For a digital decoder this number is exponential with the number of bits, which of course is much better. The neural network has the advantages to function as an associative memory.

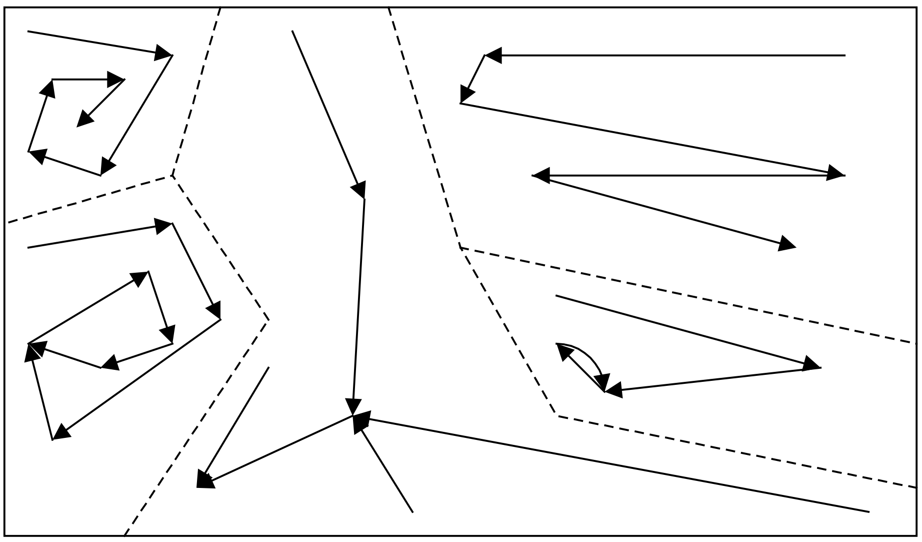


Figure 6-4 *Illustration how different states from the same attraction basin lead to the same fundamental memories*

Until now we have discussed how a network from an initial state associate with a fundamental memory. This is called auto-associativity. To illustrate the real associative memory we have to change the interpretation of the neurons. The goal is to present data to the network, which gives back some other data. More scientifically we can say that the network is associating an output with an input. Sometimes this is called hetero-associativity.

Now we divide the neurons into output neurons and input neurons. Let the neurons x_1 and x_2 represent the output and the neurons x_3 to x_5 represent the input. From the beginning the output is unknown. Therefore the initial values are set randomly. It is important that you must allow the input to change within time. If the weights are well chosen not only the output will appear at the neurons x_3 to x_5, but also the input will be filtered from noise. So if the input is a black and white picture and the value of every pixel is the input on the input neurons, the output neurons could contain information about this person e.g. address and telephone number. After the convergence the input neurons would show a filtration of the original picture.

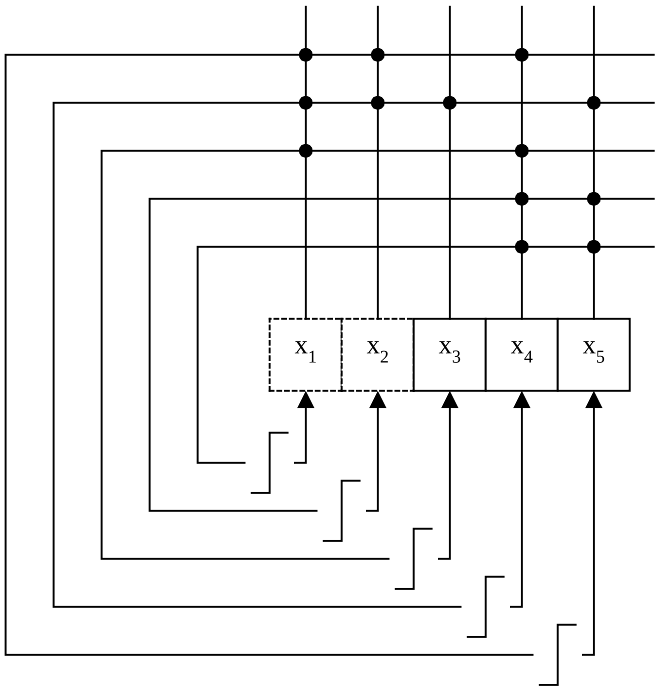


Figure 6-5 *The neurons x_1 and x_2, from figure 6-2 now illustrate the output.*

To go another step further we introduce neurons that neither are input or output. We call them hidden neurons. If we organise all the input neurons in one layer and one or more layers of hidden neurons and finally a layer of output neurons we can illustrate it in Figure 6-6. The result is then a recurrent neural network.

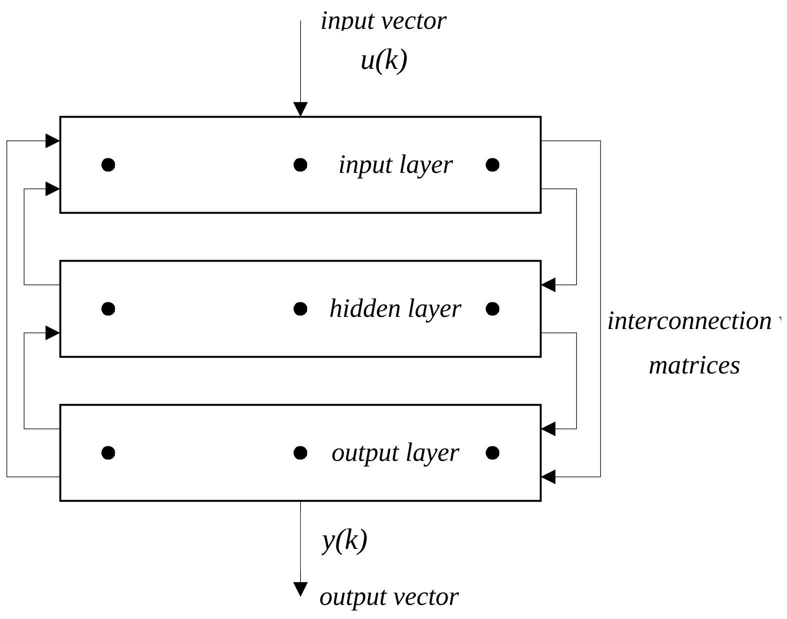


Figure 6-6 *Connections between the different layers in a multi-layer recurrent neural network. Note that the input and output are 1x3 vectors. The other arrows are 3x3 matrices containing the signals between the layers. The black dots are representing the neurons.*

6.3. Feedforward neural networks

So far there are no limitations or actual differences from the first general description in chapter 6.2. We can create a network in which the only connections allowed are from the input neurons to the first hidden layer, from one hidden layer to the next and from the last hidden layer to the output layer. If we decide that the weights are only working in the direction input towards output we have a multi-layer feedforward network. In Figure 6-7 is a scheme made to illustrate an example of a multi-layer feedforward network. Multi-layer feedforward networks always converge due to the fact that the states of the layers are calculated one by one.

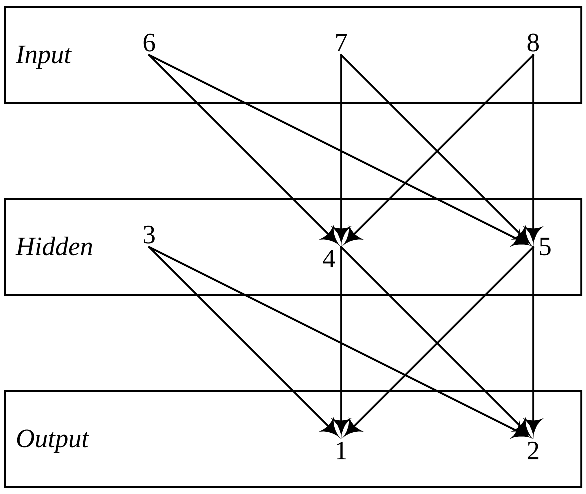


Figure 6-7 *An example of a multi-layer feedforward neural network, with the neurons 3 and 6 clamped.*

Multi-layer feedforward networks learn by examples. If the network is exposed to inputs when the wanted corresponding output is known the weights can be calculated. This method is called supervised learning. The weights are successively made better from random initial values.

Even more general upgrade rules than eq. 6-1 can be used.

$$x_i(k+1) = f\left(\sum_j w_{ij} x_j(k)\right), \mathrm{i} = 1,2,\ldots,\mathrm{n} \qquad \textbf{6-2}$$

where *f* is any arbitrary function. Normally it is required that they are limited within [–1 1] are monotonic and that have a real derivative. The most used function is probably the hyperbolic tangent function or tan-sigmoid as it is also called. To make things easier it is often written tansig. When the dynamic range 0 to 1 is chosen the logarithmic-sigmoid function, logsig, is often used. Even a linear transfer function can be useful.

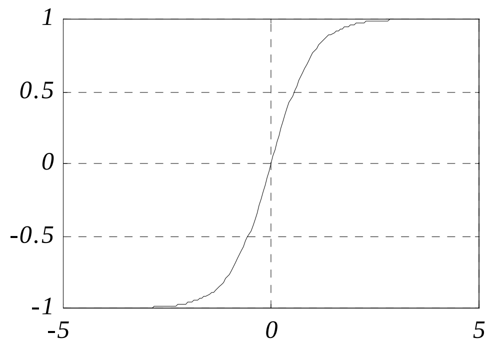


Figure 6-8 *The tansig function*

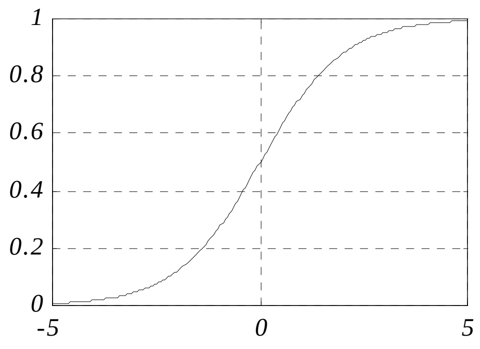


Figure 6-9 *The logsig function*

The state of some neurons can be held constant. We say that the states are clamped. The effect is that the transfer function is moved either to the left or to the right in a graph. Sometimes a threshold effect is wanted.

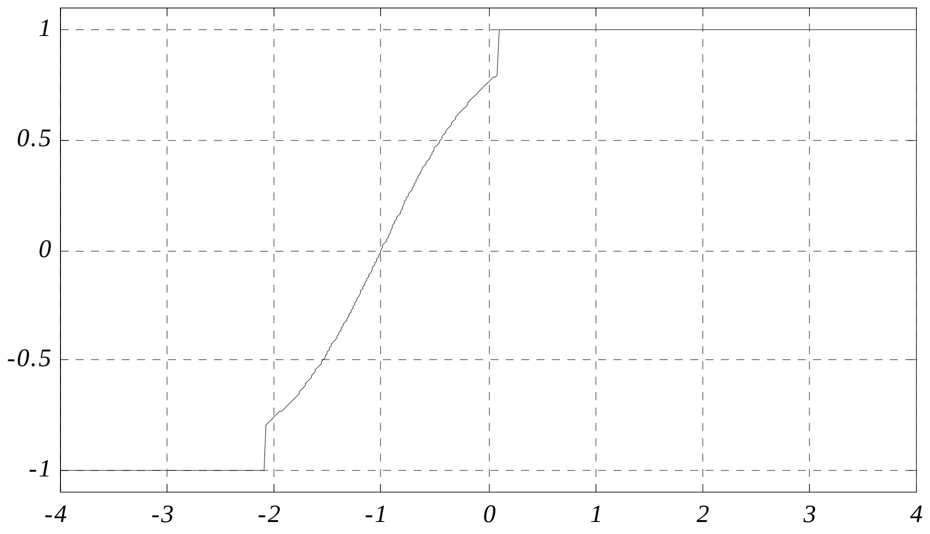


Figure 6-10 *The tansig function. This time is it connected to a clamped neuron, why it is moved one unit to the negative side. A threshold effect has been added to the function as well.*

Recurrent feedforward neural networks are a category of neural networks that are based on the feedforward neural networks discussed above. The difference is that feedback is allowed between the layers. In this report a recurrent neural networks is used.

6.4. Backpropagation

Now it is time to go a little bit deeper into how to choose the weights of the multi-layer feedforward or recurrent neural networks. The network will learn to associate a given output with a given input by adapting its weights. For this purpose the steepest descent algorithm for minimising a nonlinear function is used. For neural networks this is called backpropagation.

6.4.1. Error definitions

To describe the learning we need to define the error for a network layer (including the output layer) with *n* neurons.

$$\mathbf{E} = \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix} = \begin{pmatrix} (x_1 - \tilde{x}_1)^2 \\ (x_2 - \tilde{x}_2)^2 \\ \vdots \\ (x_n - \tilde{x}_n)^2 \end{pmatrix} \qquad \textbf{6-3}$$

If **x** is the correct output vector and the actual output is $\tilde{\mathbf{x}}$ the output error *e* is a function of the input data and the weight matrix:

$$E = [x_1 - \tilde{x}_1]^2 + [x_2 - \tilde{x}_2]^2 + \ldots + [x_n - \tilde{x}_n]^2 \qquad \textbf{6-4}$$

This is a nonlinear function in *W*, due to the nonlinear transfer function. The learning is basically a problem of minimising this nonlinear function that also is called the loss function.

The error for the hidden neurons is defined as:

$$e_i = \sum_{j=1}^{n} e_j \left(\boldsymbol{sec}\, h^2 o_j\right) w_{ji} \qquad \textbf{6-5}$$

The function sech^2 (*x*) is the hyperbolic secant that is the derivative of the tansig function. e_i is the error for neuron *i* and *o* is the state of neuron *i* before the application of the tansig function,

$$o_i = \sum_{j=1}^{n} w_{ij} x_j(k)$$

6.4.2. Updating the parameters by using backpropagation

To illustrate the training of a network using backpropagation an example is used. See Figure 6-11.

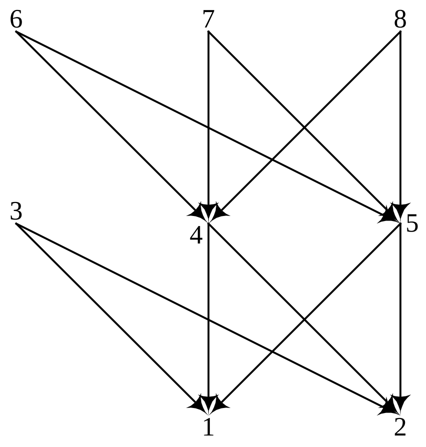


Figure 6-11 *In the example of the training this structure of net is used.*

Assume that the states x_k of the neurons can be any number [-1 1] and that the neurons 3 and 6 are clamped to 1. Also assume that the weights are collected in the matrix W and the upgrading rule is:

$$x_i(k) = \tanh \sum_{j=1}^{8} w_{ij} x_j(k-1)$$

The initial states and weights are randomly guessed. It is important this initial guess is not 0. If so there is a risk that the neuron will be stuck at this state.

Then an input is applied to the net and the states are changing according to the upgrade rule. The error is then calculated for the output layer and the weights are modified to minimise the error according to the following equation:

$$\Delta w_{ij}(k) = \eta e_i \left(\sec h^2 o_i\right) x_j$$

The derivate should not be too small due to the small changes in the weights. If the derivative is too small it is possible to add a small number to it. One suggestion is 0.05. The gain η is called the learning rate. A small value (0.001) might cause a too slow change to the weights and a big value (10) might change the value too much.

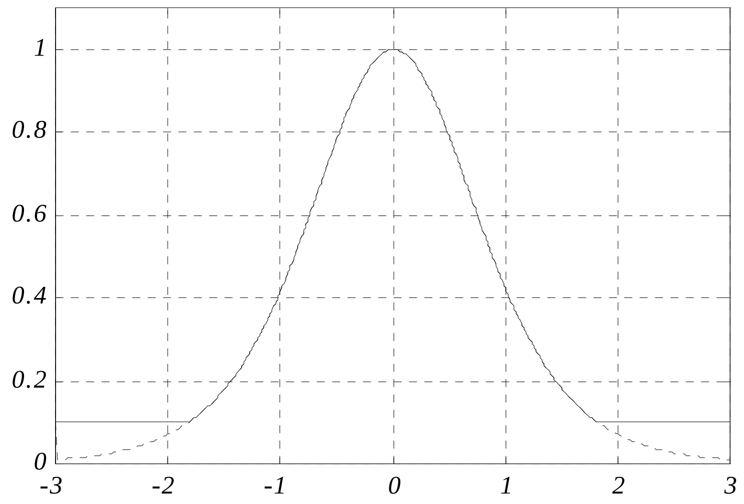


Figure 6-12 *The derivative of the tansig function. To avoid too small changes in the weights a lower limit has been introduced.*

With these new weights it is possible to calculate the errors for the hidden neurons according to equation 6-5, and the weights between the input layer and the hidden layer is updated. If there had been several hidden layers the updating would have been done one layer at the time starting with the one closest to the output.

Then another input-output pair is applied to the net and the procedure is repeated once more.

This description to adapt the weights once according to a series of input-output pairs is called an epoch. The training is repeated through several epochs to make the weights better and better. It is not known how many input-output pairs it is possible to show to a neural network but if it seems impossible to train a net to sufficient small errors, it might be because of a too small number of neurons. Also it is not known how many input-output pairs it is needed to train a neural network sufficiently.

6.4.3. The backpropagation algorithm

One way to sum up the backpropagation algorithm is:

$$\Delta w_{nj} = -\eta_k \frac{\partial E}{\partial w_{nj}}$$

where Δw_k is the change of the weight, $\frac{\partial E}{\partial w_{nj}}$ is the current gradient vector of the update functions and η is the learning rate.

6.5. Methods for adjusting the adaptation gain

6.5.1. Quasi-Newton algorithms

Even though the error function is decreasing fast in the direction of the negative of the gradient it is not sure that it will lead to the fastest convergence. The conjugate gradient algorithms also calculate the learning rate for the different iterations, so that the loss function is minimised along the negative gradient. The performance index is the size of the error along the negative gradient. There are many different methods to choose η. One group of methods is the Quasi-Newton algorithms. The basic upgrade rule is:

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \mathbf{H}^{-1}(k)\nabla\mathbf{E}(k) \qquad \textbf{6-6}$$

where **H**(k) is the second derivatives of the performance index. The name of this matrix is the Hessian matrix. It is important that **H**(k) has an inverse.

6.5.2. The Levenberg-Marquardt algorithm

The Quasi-Newton methods usually converge faster than ordinary conjugate gradient rules, but are complex and expensive to compute. To avoid the computation of the second derivative the Levenberg-Marquardt algorithm approximates the Hessian matrix by:

$$\mathbf{H} = \mathbf{J}^T\mathbf{J} \qquad \textbf{6-7}$$

and the gradient can be computed as:

$$\nabla\mathbf{E} = \mathbf{J}^T\mathbf{e} \qquad \textbf{6-8}$$

where **J** is the Jacobian matrix which contains of the derivatives of the network errors with respect to the weights. The Jacobian matrix is much less complex to compute than the Hessian matrix. The update is then as follows:

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \left[\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}\right]^{-1}\mathbf{J}^T\mathbf{e} \tag{6-9}$$

where μ is a parameter that is decreased at every successful iteration. It is only increased when the calculated change would increase the loss function. Another important aspect of μ is that it makes sure that the matrix has an inverse.

The problem with the Levenberg-Marquardt algorithm is that it contains storage of matrices that can be quite large. The size of the Jabobian matrix is $Q \times n$, where Q is the number of training sets and n is the number of weights and biases in the network.

6.6. Simulations

To check the validity of the model created it is of course possible to make simulations. There are two different types of possible simulations. The operation modes are called parallel and serial mode simulations. In the parallel simulation the available outputs from the past come from the output of the neural model and in the serial those old outputs come from the real process.

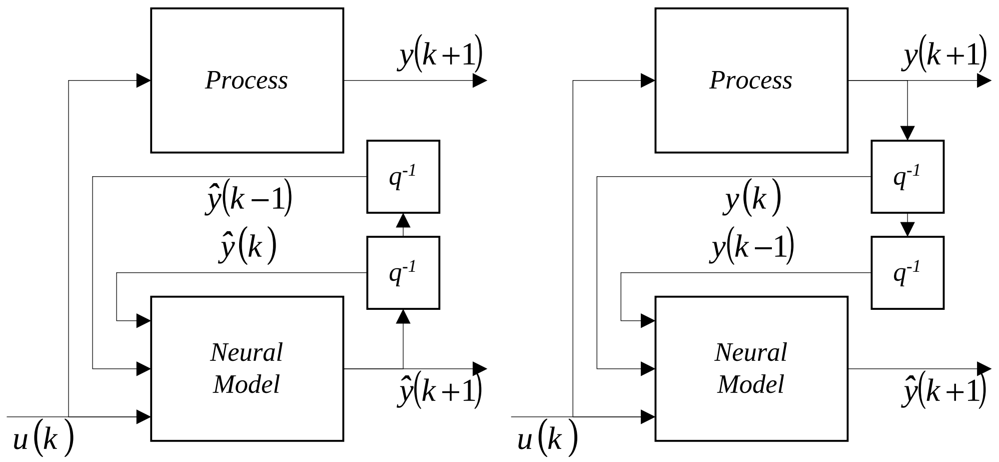


Figure 6-13 *Simulations in two different ways. To the left: Parallel simulation that uses the output from the neural network for its prediction. To the right: Serial simulation that uses the output of the real process to predict the next output.*

6.7. Architecture for the neural network

Now is the time to use the network for control applications. Once again there are numerous approaches for this task. For example there are networks that are containing both the controller and the identification part of the control system. In this work the neural network is only a model of the process. The regulator is here not a neural network. There are many different writers that have suggested their solutions. It is important at this point to remember that the different notations used in the previous sections, inside the neural nets might change its meaning. Here is a system containing the normal linear regulator with a neural nonlinear process model. (Henriques et al, not yet published) suggested the network architecture, with two input neurons, three neurons in the hidden layers and one output neuron.

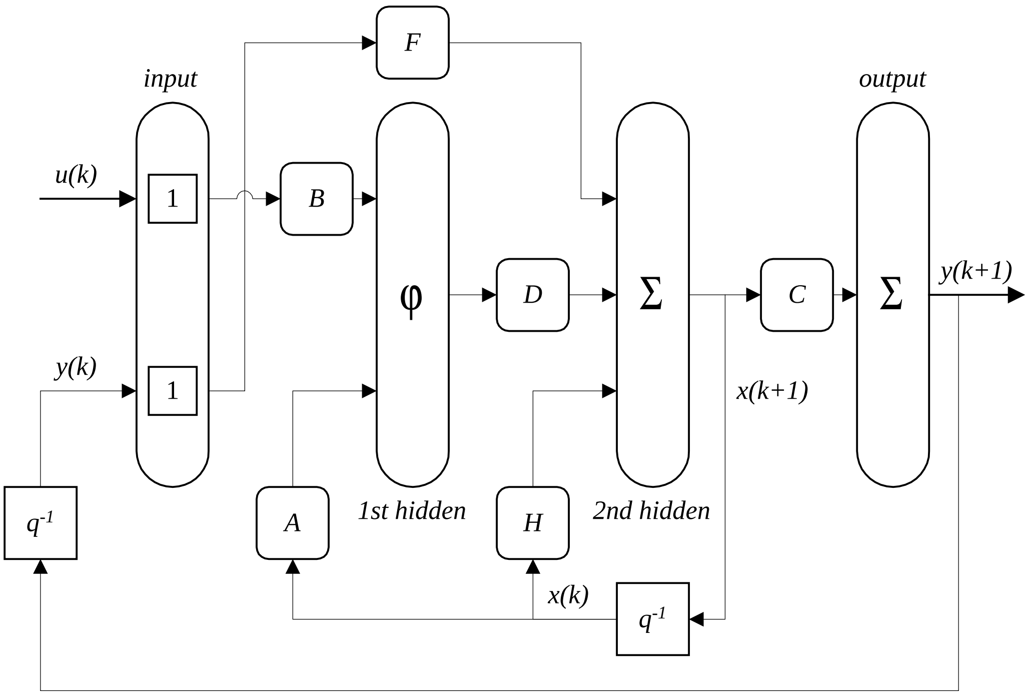


Figure 6-14 *Neural network architecture that is used in this work. φ is here the tansig function.*

The equations can be described as follows:

$$\begin{cases} x(k+1) = D\varphi\{Ax(k) + Bu(k)\} + Hx(k) + Fy(k) \\ y(k) = Cx(k) \end{cases} \qquad \textbf{6-10}$$

The vector $x(k)\in \Re^n$ is the output from the output of the second hidden layer, $u(k)\in \Re^m$ is the input and $y(k)\in \Re^p$ is the output. The matrixes *A*, *B*, *C*, *D*, *F* and *H* are interconnection matrixes, containing the weights. *A*, *D* and *H* have size 3x3, *B* and *F* have size 3x1 and *C* has size 1x3. *D* is clamped to be the identity matrix. In all this means 27 parameters to choose in comparison to 4 in the linear model. It is obvious that it is a much more complex task to choose that many parameters.

The final result is then:

$$\begin{cases} x(k+1) = D\varphi\{Ax(k)+Bu(k)\}+\{H+FC\}x(k) \\ y(k) = Cx(k) \end{cases} \qquad \textbf{6-11}$$

It is theoretically difficult to explain why the vector *F* is necessary. *H* is enough to accomplish full freedom for the expression *H+FC*.

6.8. Input-output feedback linearisation control

A nonlinear function ψ is put before the nonlinear process, according to the figure below, so that the transfer function *v*→*y* is linear.

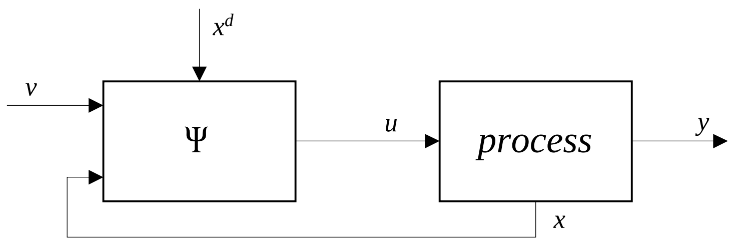


Figure 6-15 *The general idea of input-output feedback linearisation is to use a nonlinear function Ψ put before the process making the function y=f(v) linear.*

$$u = \Psi\left(x, x^d, v\right)$$

where *x* and x^d represents the state vector information from the process and from the desired resulting system, respectively.

Assume that the process is modelled with a state-space description:

$$\begin{cases} x(k+1) = \boldsymbol{f}\left(x(k), u(k)\right) \\ y(k) = \boldsymbol{h}(x(k)) \end{cases} \qquad \textbf{6-12}$$

where x is the state vector and y is the output from the system. f and h are both smooth functions.

Further on we define the discrete decoupling matrix as:

$$\mathbf{E}(\mathbf{x}(k), \mathbf{u}(k)) = \frac{\partial}{\partial \mathbf{u}(k)} \{\mathbf{y}(k+r)\} \qquad \textbf{6-13}$$

where $x \in \Re^p$ and $y \in \Re^p$.

The desired system is described as state space model:

$$\begin{cases} x^d(k+1) = Ax^d(k) + Bv(k) \\ y^d(k) = Cx^d(k) \end{cases} \qquad \textbf{6-14}$$

$\boldsymbol{\Psi}$ is obtained by solving the following equation in respect to $\mathbf{u}(k)$:

$$\mathbf{E}(x(k), u(k)) = CA^r x^d(k) + CA^{r-1}Bv(k) + CA^{r-2}Bv(k+1) + \ldots + CBv(k+r-1) \qquad \textbf{6-15}$$

The output will then be:

$$y(k+r) = CA^r x^d(k) + CA^{r-1}Bv(k) + CA^{r-2}Bv(k+1) + \ldots + CBv(k+r-1) \qquad \textbf{6-16}$$

6.8.1. Exact calculation of the feedforward linearisation

To solve the equations 6-16 and 6-17 there are two main methods, one exact method and one approximate method. The exact method can only be used in rare special cases. If the output signal can be described as:

$$y(k+1) = f(x(k)) + G(x(k))u(k) \qquad \textbf{6-17}$$

the system is called affine and is possible to solve exactly if the matrix *G(x(k))* is invertible.

If the control law is chosen as:

$$u(k) = p(x(k), x^d(k)) + Q(x(k))v(k) \qquad \textbf{6-18}$$

and

$$\begin{aligned} p(x(k), x^d(k)) &= G^{-1}(x(k))\{- f(x(k)) + CAx^d(k)\} \\ Q(x(k)) &= G^{-1}(x(k))CB \end{aligned} \qquad \textbf{6-19}$$

the result is the MIMO (*p x p*) linear system described by:

$$y(k+1) = CAx^d(k) + CBv(k) \qquad \textbf{6-20}$$

6.8.2. Approximate solution of the feedback linearisation

One possible method to solve eq. 6-16 and 6-17 is based on a linearisation at a working point. In this method the description of the output is:

$$y(k+1) = f(x(k), u(k)) \qquad \textbf{6-21}$$

The linear approximation, with the help of a Taylor's series the following is obtained:

$$y(k+1) = f_0 + \nabla F_0 \Delta x(k) + \nabla E_0 \Delta u(k) \qquad \textbf{6-22}$$

f_0 is the calculated operating point and ∇F_0 is containing the partial derivatives of *f* with respect to *x(k)* and ∇E_0 is containing the partial derivatives with respect to *u(k)*.

The control law is the same as with the exact calculations. See eq 6-18.

From this it is possible to calculate the feedback linearisation control law as:

$$\begin{aligned} p(x(k), x^d(k)) &= u(k-1) + \nabla E_0^{-1}\{- f_0 - \nabla F_0 \Delta x(k) + CAx^d(k)\} \\ Q(x(k)) &= \nabla E_0^{-1} CB \end{aligned} \qquad \text{6-23}$$

This results in a system output described as:

$$y(k+1) \approx CAx^d(k) + CBv(k) \qquad \text{6-24}$$

This is if the higher order terms of the Taylor´s series can be neglected and *∇E₀* is non-singular.

In the experiments used for this work we have the following model (eq. 6-11):

$$\begin{cases} x(k+1) = D\,\boldsymbol{tanh}(Ax(k) + Bu(k)) + (H + FC)x(k) \\ y(k) = Cx(k) \end{cases}$$

From this follows that:

$$y(k+1) = CD\,\boldsymbol{tanh}(Ax(k) + Bu(k)) + C(H + FC)x(k)$$

$$\nabla F_0 = CD\left(\left(\frac{\partial\,\boldsymbol{tanh}(x(k))}{\partial x}\right)A + CH + CFC\right)$$

and

$$\nabla E_0 = CD\left(\left(\frac{\partial\,\boldsymbol{tanh}(x(k))}{\partial u}\right)B\right)$$

The control signal is calculated as:

$$\begin{aligned} u(k) &= p\left(x(k), x^d(k)\right) + Q(x(k))v(k) = \\ &= u(k-1) + \nabla E_0^{-1}\left\{-f_0 - \nabla F_0 \Delta x(k) + CAx^d(k)\right\} + \nabla E_0^{-1} CBv(k) = \\ &= u(k-1) + \nabla E_0^{-1}\left\{-f_0 - \nabla F_0 \Delta x(k) + CAx^d(k) + CBv(k)\right\} \end{aligned} \tag{6-25}$$

6.8.3. Linear predictive control

Assume that the whole system can be described as in Figure 6-16. (Braake, 1997 & 1999), suggested this model. This kind of control is called linear predictive control and is here used together with the input-output linearisation.

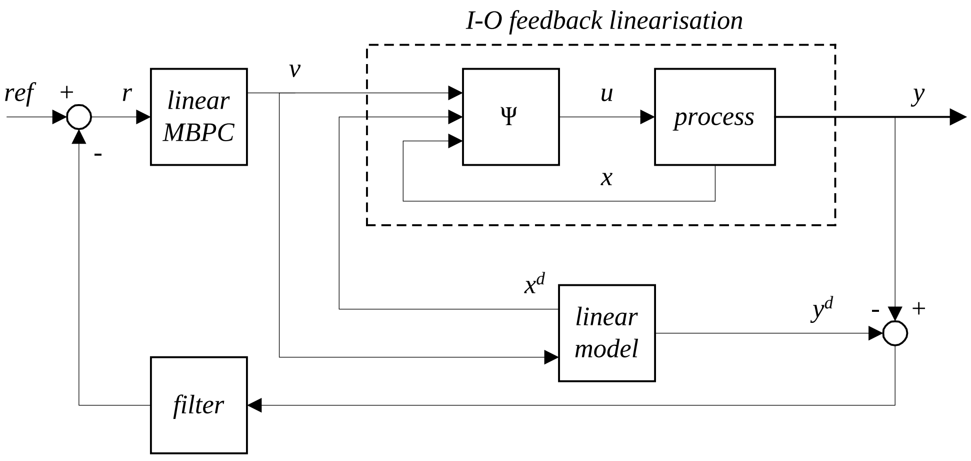


Figure 6-16 *Block diagram showing the entire input-output feedback linearisation.*

The model-based predictive control, MBPC, is used to optimise the control. The loss function to minimise is:

$$J(\tilde{v}) = (\tilde{y} - \tilde{r})^T (\tilde{y} - \tilde{r}) + \Delta\tilde{v}^T W_v \Delta\tilde{v}$$

where W_v is a square positive definite diagonal weighting matrix of the controller outputs.

$\tilde{y}$ can then be modelled as:

$$\tilde{y} = R_x x_k^d + R_u \tilde{v}$$

where R_x and R_u are matrices.

This is optimised when $R_x = CA$ and $R_u = CB$. (See (Braake, 1997 & 1999)). The result is that the reference of the system can be described as:

$$r(k) = CAx^d(k) + CBv(k)$$

Due to the fact that we are dealing with a SISO system, the following control law is:

$$u(k) = u(k-1) + \frac{-y(k) - \nabla F_0 \Delta x + r(k)}{\nabla E_0}$$

To get better control behaviour a pole, *a*, is introduced.

$$u(k) = u(k-1) + \frac{(a-1)y(k) + (1-a)r(k) - \nabla F_0 \Delta x}{\nabla E_0} \qquad \textbf{6-26}$$

7. Experimental results with the neuro control

7.1. Identification

The neural network suggested by Henriques (Henriques *et al*, *not yet published*) was trained with the Levenberg-Marquardt algorithm in Matlab. See appendix 8.2 and 8.3. The same open loop data that was the base for the linear identification was used. The input was a signal with 250 samples, and the training was made over five epochs. This means that the network was exposed to 1250 input-output pairs. Figure 7-1 shows the parallel and serial simulations of the neural network model. The upper signal is the output from the real process and the signal furthest down is the parallel simulation and the signal in the middle is the serial simulation. See Figure 6-13 for definitions of the different simulations.

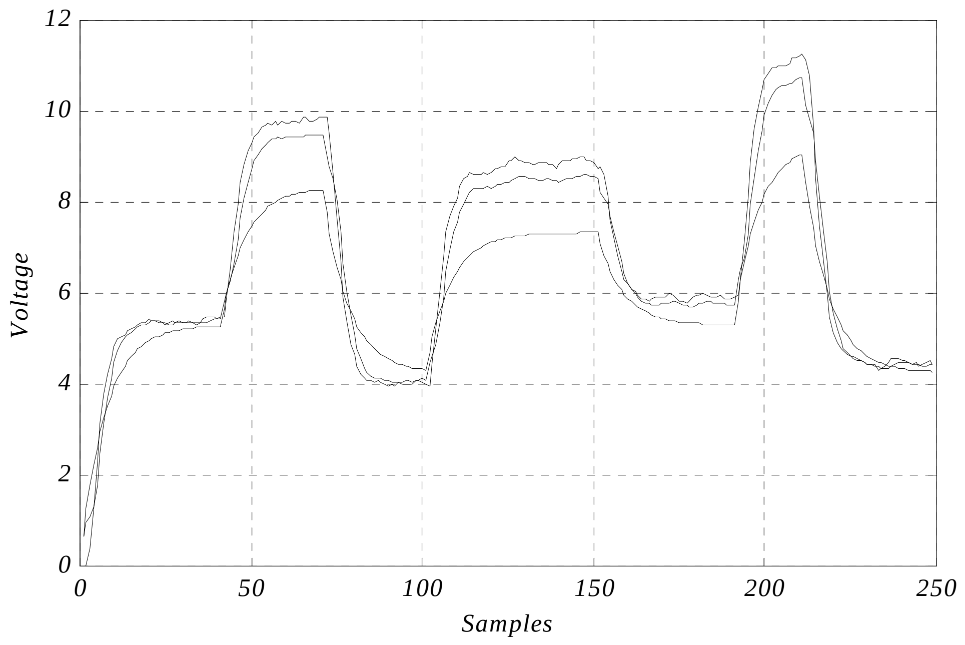


Figure 7-1 *The simulation result with the neural network model of the process. At the top the real system. In the middle the serial simulation and at the bottom the parallel simulation.*

In the linear case the simulations are made with the knowledge of the real output. The simulation method best suited for comparison is therefore the serial simulation. With this in mind it possible to see that the result is quite satisfying. The model is corresponding to the real process quite well. It is not very fair to compare a linear and a nonlinear model when the process is almost perfectly linear. To get a better model with the neural network than with a linear model, the process has to be quite strongly nonlinear. For the PT326 process this is not the case.

7.2. Control

7.2.1. Simulations

The result of the control simulations can be seen in Figure 7-2. The reference is the square signal and the output is slightly above the neural network estimation. There is no steady state error nor over-shoots and still the control is fast. There is noise added to the simulation to create a small error for the neural network.

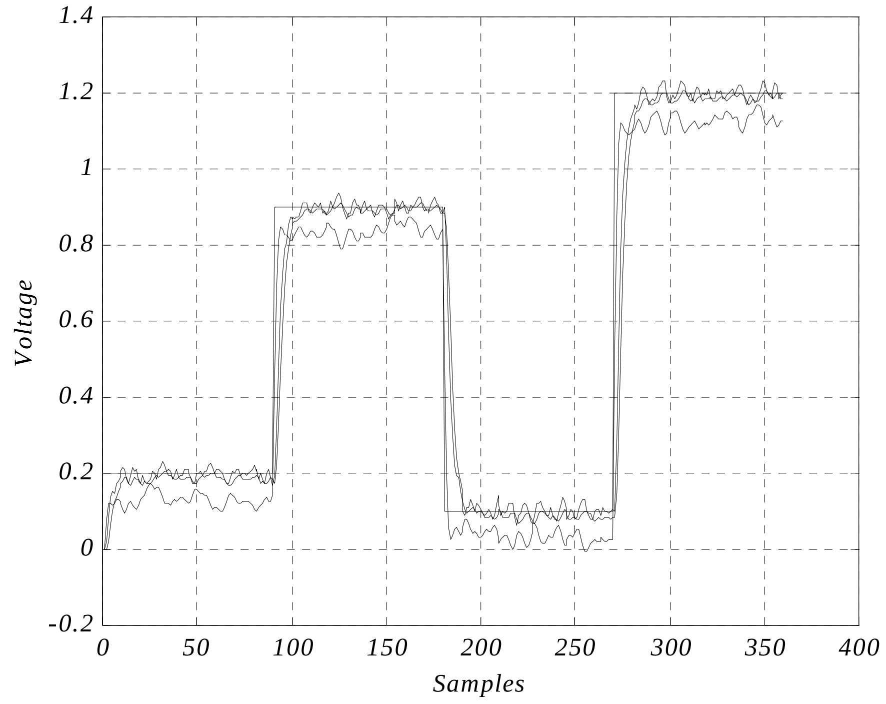


Figure 7-2 Simulation of the control using the input-output feedback linearisation. The neural network prediction and the output are both very close to the reference. The control signal is a little bit further down. To the output there is added noise.

7.2.2. Real process

With the first attempt with the feedforward linearisation the control was unstable. The regulator compensates the error like a dead-beat regulator. To avoid big variations of the control signal the control law was changed a bit:

$$u(k) = u(k-1) + \alpha \frac{(a-1)y(k) + (1-a)r(k) - \nabla F_0 \Delta x}{\nabla E_0} \tag{7-1}$$

where α has a small value e.g. 0.05.

With α=0.05 and the pole a=0.6 the result was the following (Figure 7-3):

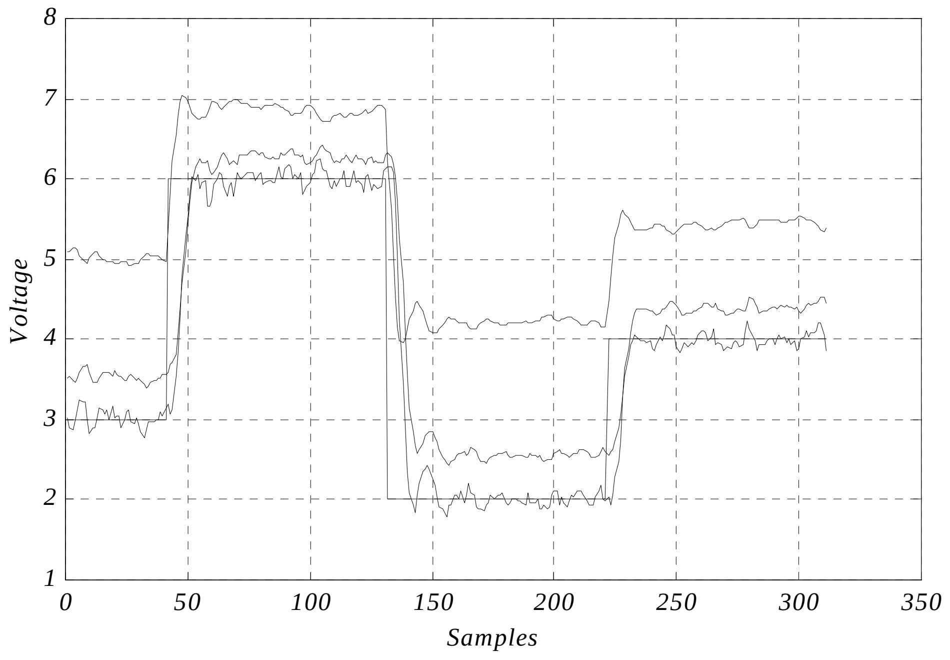


Figure 7-3 *The result of the control using input-output feedback linearisation. The reference signal is a couple of steps; the upper curve is the control signal. The output is following the reference well and the predicted output from the neural net is situated between the output and the control signal.*

The reference is rectangular, the output is following well. Slightly above is the estimation is slightly to big and the control signal is situated on top.

If the inlet of air was decreased by closing the opening to a minimum the result was not so good (See Figure 7-4).

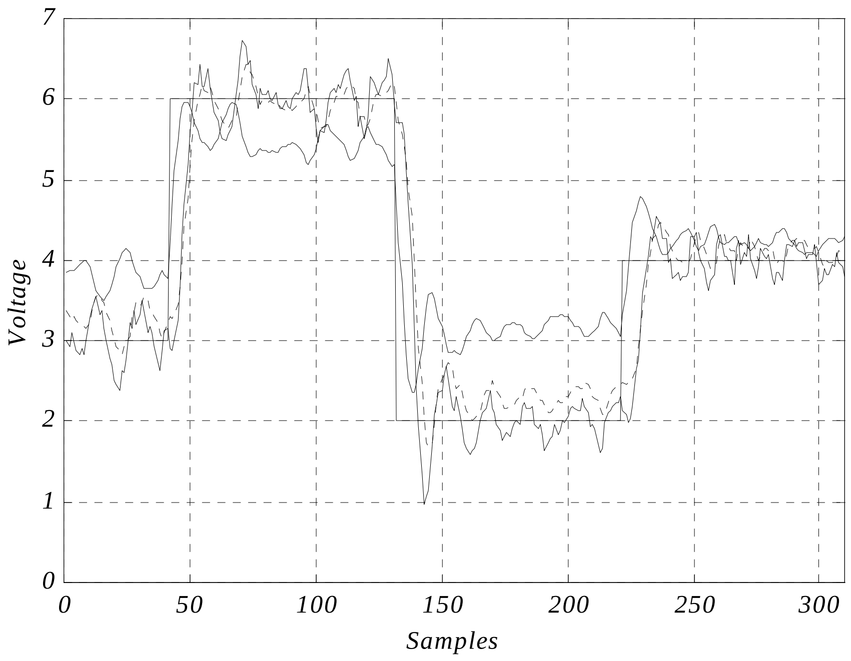


Figure 7-4 *The control using input-output feedback linearisation with a load on the process.*

It did not help if a slower pole was chosen. An adaptive identification of the process could have been to a great help. The general idea for the identification would be by identifying the matrices in eq. 6-12: *A, B, C* and *H+FC,* with the normal adaptive methods. This means a lot of parameters and is most likely very difficult but should be possible. Some other approaches would be keeping some parameters constant e.g. the parameters *A* and *B*.

7.3. Conclusions

The result was quite good with the input-output feedforward linearisation control. To generalize a little bit, to see what neural networks have to give to the automatic control field the most important is to evaluate the neural network part of the control. The feedforward linearisation techniques can be used with any non-linear model, not only neural networks.

It is difficult to make a good identification with the neural networks. They might look very nice in the theory but to create a good model can more easily be made in other ways. The PT326 process is almost linear. That makes is slightly unfair to compare the identification made with the neural net and the linear model. If the process had been nonlinear it might be possible to see advantages with the neural network. It is possible to see that the recurrent network makes the linear identification well, just as well as the identification made in chapter 3. The conclusions are that if the process is nonlinear the neural network approach might be better than the linear approximation the linear control provides. This has to be verified by further studies before it is considered as a fact. It is important to notice that there are other ways to build nonlinear regulators than with neural networks.

The fact that changes of the parameters change the behaviour quite much is worrying. It seems possible but difficult to create an adaptive on line identification of the process parameters.

8. Appendix

8.1. Matlab code for the adaptive control

```
function main

%------------------------ Initialization

initdac;                %the data aqusition board is initialized

Amod=[ 1  -1.5 0.5 ];   %The initial parameters of the system model
Bmod=[ 0  0.2  0.5 ];

ord=2;                  %the order of the system is initialized

Acl=zeros(1,ord+1);     %The closed loop shold have this size
Acl(1:ord+1)=poly([ 0.85 0.8 ]);    %The closed loop poles are...
Ts=200;                 %Sampling time

F = 3*eye(4);           %The covariance matrix is initialized
Th= [Amod(2) Amod(3) Bmod(2) Bmod(3)]’;   %The parameter vector is
                                          %initialized
lamb0=0.97; lamb1=0.95; lamb2=1;          %Forgetting factors are
                                          %initialized
alpha=lamd1/lamb2;

uk=3*sum(Amod)/sum(Bmod);     %The initial value are chosen so the
                              %output is at
                              %steady state with the value 3

uk1=uk; uk2=uk; uk3=uk; uk4=uk; uk5=uk;   %and the ’former’ values
                                          %are initialized according
                                          %to this
yk=3; yk1=yk; yk2=yk;
ykf=yk; ykf1=yk; ykf2=yk; ykf3=yk;

%------------------------------- Define input

I = ([3*ones(1,60) 5*ones(1,48) 2*ones(1,72) 4*ones(1,41)
1.5*ones(1,60) 5*ones(1,90) 2*ones(1,60)]);
                                          %a input vector is created
N=size(I);                                %N = the input length
N=N(1,2);

AB=zeros(N,4);                %AB is a matrix containing all the
                              %model parameters throughout time
lamb=zeros(N,2);              %lamb is a matrix containing the
                              %forgetting factors throughout time

Up=zeros(N,1);                %control signal vector is initialized
Yp=zeros(N,1);                %output vector is initialized

%---------------------------- Diophantine equation

[S,R,T]=colpolos(Bmod,Amod,Acl,ord,1);    %the Diophanting
                                          %equation i solved.
                                          %Normally it is not needed
                                          %at this point
                                          %but here it is needed to
                                          %avoid errors

%------------------------------ control loop -----------------------

for k=1:N

   st=myclock;          %the clock is ’started’
   rk=I(k);             %the reference is I(k)

%----------------------- output
   yk = read;                 %a new output is read from the process
   ykf=0.6*yk+0.4*ykf1;       %the feedbacked output is filtered

   %yk = B(1:3)*[uk uk1 uk2]' -A(2:3)*[yk1 yk2]';     %the simulated
                                                %output is calculated
                                                %if wanted
%---------------------------- update adaptive parameters

   o  = [-ykf1 -ykf2 uk1 uk2]';     %the old-values-vector is updated
   ek = (ykf - Th' * o)/(1 + o' * F * o); %the error is calculated
   Th = Th + F * o * ek;      %The new parameters are calculated

   %lamb1=lamb0*lamb1+1-lamb0;      %the forgetting factors are
   %lamb2=1;                        %udated according to some rule


   lamb1=trace(F-((F*o*o'*F)/(alpha+o'*F*o)))/trace(F);
   lamb2=lamb1/alpha;

   F  = (1/lamb1)*(F - (F*o*o'*F)/((lamb1/lamb2) + o' * F * o));
                              %the covariance matrix are updated

   Amod=[1 Th(1:2)'];                     %the new model is extracted
   Bmod=[0 Th(3:4)'];

%---------------------------- Diophantine equation

   [S,R,T]=colpolos(Bmod,Amod,Acl,ord,1); %the Diophantine is solved

%----------------------------- Some data is saved

   AB(k,:)=[Amod(2:3) Bmod(2:3)];
   lamb(k,:)=[lamb1 lamb2];
   err(k)=ek;

%----------------------- control

   uk = T*rk - R(2:ord+1)*[uk1 uk2]' - S*[ykf ykf1 ykf2]';
                                  %the control is calculated

   uk=0.6*uk+0.4*uk1;
                                  %... and filtered

   if uk>10, uk=10.0; end;        %The control is limited
     if uk< 0, uk= 0.0; end;
   write(uk);     %the control is put at the input of the process
   disp([ rk yk uk ])        %The signals are written on screen
```

```
%----------------------- update states
   Yp(k)=yk;
   Up(k)=uk;

   yk2=yk1; yk1=yk;
   uk5=uk4; uk4=uk3; uk3=uk2; uk2=uk1; uk1=uk;
   ykf3=ykf2; ykf2=ykf1; ykf1=ykf;

   et=myclock;                            %the clock is ’stopped’

   if (et-st)<Ts & (et-st)>=0       %the system is delayed so the
      delay(Ts-et+st);              %sampling time is correct
   else
      disp(’ ------ short sampling time -----’)
   end ;

%--------------------------- control loop end ------------------

end


savedat(’ab.dat’,AB)                %the data is saved to files
savedat(’lamb.dat’,lamb)
dat=[I’ Yp Up];
savedat(’data.dat’,dat)
```

8.2. Matlab code for identification of the neural network

```
%                                                   DATA
%                                                ============

load ur.dat
load yr.dat
filt;

inp= ur';
out= yr';


inicio= 1;
fim = length(out)-1;
TAM = fim-inicio+1

UR=  inp(:,inicio:fim);
YR=  out(:,inicio:fim);
YR=YR-YR(1);
YR1=  YR;
for i=1:TAM-1                      %YR1 is YR shifted one step to
      YR1(:,i+1)=YR(:,i) ;         %the right (same length)
end
UR1=  UR;
      for i=1:TAM-1   UR1(:,i+1)=UR(:,i) ;
end

plot([YR' UR' YR1'])

%                                                  INITIAL
%                                                ============
nu = 1;                                  % # inputs =nu
ny = 1;                                  % # outputs =ny
nx = 3;                                  % # neurones in layer x=nx
nz = 3;                                  % # neurones in layer z=nz
EPOCAS =  4                              % # epochs =epocas
EXTERN =  0                              %

TREINA =  input('TREINA =  ') % help variable TREINA is chosen. 1
                              %means train, 0 means not train

to=clock;                                %start time
ss_mod                                   %run the function ss_mod
tempo=etime(clock,to)/60                 %the run time is calculated
```

8.3. Matlab code for initialisation, training and simulating the neural net

```
if TREINA
   %                                    NETWORK DEFINITON
   %                                    =================
numInputs = 2;    %# inputs to the net are 2. (input and feedback)
numLayers = 3;    %# layers are 3;

tnet = network(numInputs,numLayers);      %the net is created with
                                          %the right amounts of
                                          %inputs and outputs
tnet.biasConnect   = [0 0 0 ]';           %bias = zero;
tnet.inputConnect  = [1 0 ; 0 1 ;0 0 ];   %define connections for the
                                          %weights between the layers
tnet.layerConnect  = [0 1 0 ; 1 1 0; 0 1 0];
tnet.outputConnect = [0 0 1 ];
tnet.targetConnect = [0 0 1 ];

tnet.inputs{1}.size = nu;           %#inputs to the first input
                                    %node are nu from INITIAL
tnet.inputs{2}.size = ny;
                                    %#inputs to the second input node
                                    %(feedback) are ny from INITIAL
tnet.inputs{1}.range = [ -1*ones(nu,1) 1*ones(nu,1)];
                                    %the input range are defined
tnet.inputs{2}.range = [ -1*ones(ny,1) 1*ones(ny,1)];
tnet.layers{1}.size = nz;           %#nodes in layer 1 is...
tnet.layers{1}.transferFcn = 'tansig';    %#output function from
                                          %layer 1 is...
tnet.layers{2}.size = nx;                 %#nodes in layer 2 is...
tnet.layers{2}.transferFcn = 'purelin';
tnet.layers{3}.size = ny;                 %#nodes in layer 3 is...
tnet.layers{3}.transferFcn = 'purelin';
tnet.layerWeights{2,2}.delays = 1;  %delay in layer(.),node(.) is ...
tnet.layerWeights{1,2}.delays = 1;

WA = 0.1*rand(nz,nx);   %Parameters are ramdomly chosen exept for WD
WB = 0.1*rand(nz,nu);
WC = 0.1*rand(ny,nx);
WD = eye(nx,nz);
WF = 0.1*rand(nx,ny);
WH = 0.1*rand(nx,nx);

tnet.IW{1,1} = WB;      %the parameters are put into the net
tnet.LW{2,2} = WH;      %tnet.LW{1,2} means to layer 1 from layer 2
tnet.LW{1,2} = WA;
tnet.LW{2,1} = WD;
tnet.LW{3,2} = WC;
tnet.IW{2,2} = WF;

disp(' -------- NETWORK initialization --------- ')

%                                               NETWORK TRAINING
%                                               ================
tnet.performFcn = ’mse’;            %the parameters are changed
                                    %according to the function...
                                    %Here ’mse’=mean square error
tnet.trainFcn = ’trainlm’;          %the learning training method
                                    %is... Here Levenberg-Marquardt

tnet.trainParam.goal  = 1e-15;      %the goal of the training is to
                                    %have an error less than...
tnet.trainParam.epochs= EPOCAS;     %# epochs is... If the # of
                                    %iterations is exeeded the
                                    %training will stop
tnet.trainParam.show  = 1;          %the training status will be
                                    %shown every... iteratrions of
                                    %the algorithm
tnet.trainParam.mu_max  = 1e50;     %mu is the initial value for my.

tnet.layerWeights{2,1}.learn=0.0;   %the learning rate for node (.,.)

US = con2seq(UR1);      %makes a sequence of the shifted input signal
YS = con2seq(YR1);      %makes a sequence of the shifted output
TS = con2seq(YR);       %makes a sequence of the unshifted output
XS = [ US ;  YS ];

tnet= train(tnet,XS,TS);            %training begins

WF = tnet.IW{2,2};                  %The weights of ... are...
WB = tnet.IW{1,1};
WH = tnet.LW{2,2};
WA = tnet.LW{1,2};
WD = tnet.LW{2,1};
WC = tnet.LW{3,2};

ys  = sim(tnet,XS);                 %simulation starts
ys  = seq2con(ys);                  %... and is made into a sequence
YS  = ys{1,1};                      %makes YS to the output vector of
                                    %the simulated net

disp(’ -------- NETWORK learning --------- ’)

data3=[ UR1’ YR1’ YR’ YS’ ];

UR=UR*10;
YR1=YR1*10;
YR=YR*10;
YS=YS*10;

save data3.dat data3 /ascii   %saves data to file
save WA.dat WA /ascii
save WC.dat WC /ascii
save WD.dat WD /ascii
save WB.dat WB /ascii
save WF.dat WF /ascii
save WH.dat WH /ascii
plot([YR’ YS’])         %plots the real system with the simulation

else
%                                       NETWORK VALIDATION
%                                       ==================

load WA.dat                               %loads existing data
load WB.dat
load WC.dat
load WD.dat
load WF.dat
load WH.dat
load data3.dat
UR1= data3(:,1:nu)’;          %reference input delayed one sample
YR1= data3(:,nu+1:nu+ny)’;    %reference output delayed one sample
YR = data3(:,nu+ny+1:nu+ny+ny)’;               %reference output
YS = data3(:,nu+ny+ny+1:nu+ny+ny+ny)’;    %output from the simulink
                                          %simulation made during the
                                          %learning
[nz,nx]=size(WA);
[nx,nu]=size(WB);
[ny,nx]=size(WC);
TAM=length(YR);

YN= zeros(ny,TAM);
xn= zeros(nx,1); xn1=xn;
yn= YR1(:,1); yn1=yn;

for t=1:TAM                         %simulation
      uk1= UR1(:,t);
      yk1= YR1(:,t);
      xn = tansig( WA*xn + WB*uk1) + WF*WC*xn + WH*xn;
      yn = WC*xn ;
      YN(:,t)=yn;                   %output from the neural net and
                                    %the simulation made above
      yn1=yn;
end

YN=YN*10;
YR=YR*10;

plot([YR’ YN’])                     %plots the simulation
ERRO3 = sum(abs(YR’-YN’))           %The error is...

end
```

8.4. Matlab code for control with feedforward linearisation

```
clear
pack

%                                          DEFINITONS
%                                       ================

load WA.dat                              %Parameters are loaded
load WB.dat
load WC.dat
load WD.dat
load WF.dat
load WH.dat

[nz,nx]=size(WA);                  %Size of the layers are read
[nx,nu]=size(WB);
[ny,nx]=size(WC);
xo = zeros(nx,1);

%-------------------------------------- Pole placement controller

%-------------------------------------- Reference

T  = 1;
tf = 360;
t  = 0:T:tf-T;
TAM= length(t);
RR = referenc([  0.2  0.9  0.1  1.2  ]',TAM/4    )';
                                         %create reference values

%-------------------- real system initialization
xn=0.01*rand(nx,1); xn1=xn;
wk = 0;   wk2=wk;  wk1=wk;
uk = 0;   uk2=uk;  uk1=uk;
yk = 0;   yk2=yk;  yk1=yk;
yn = 0;   yn2=yn;  yn1=yn;
ek = 0;   ek2=ek;  ek1=ek;
vk = 0;   vk2=vk;  vk1=vk;
iek= 0;
en = 0;                                  %error with neural net
er = 0;                                  %error with 'real' system
%-------------------- neural model initialization
xn= zeros(nx,1);
zn= zeros(nz,1);
zo= zeros(nz,1);
yn= zeros(ny,1);
%-------------------- real system + neural model
dx= zeros(nx,1);
du= zeros(nu,1);
YN= zeros(ny,TAM);
YR= zeros(ny,TAM);
UR= zeros(ny,TAM);
EN= zeros(ny,TAM);
ER= zeros(ny,TAM);
WR= zeros(ny,TAM);
YN(:,1) = yn;
YR(:,1) = yk;
```

```
UR(:,1) = uk;
ER(:,1) = er;
ER(:,1) = en;

%                                          CONTROL LOOP
%                                       ==================

for time=2:TAM

      rk = RR(time) ;

      %------------------------ real system output
      yk=pt326(uk1,uk2,yk1,yk2);          % produces the output, y,
                                          %from the process
      %------------------------ neural model output
      xn1=xn;
      xn = tanh(WA*xn + WB*uk ) + WF*yk1 +WH*xn ;
                        %calculates the state, x, of the neural net
      yn = WC*xn ;      %calculates the output yl

      %-------------------------------------- linearization
      dx = xn-xn1;                        %calculates dx
      J  = diag( sech(WA*xn + WB*uk) );   %calc. the deriv. of yl

      FL = WC*(J*WA+WH+WF*WC);      %calculates the linearisation of
                                    %the neural network
      GL = WC*J*WB;                 %calculates G
      ig = inv(GL);                 %calculates the inverse of G

      %-------------------------------------- control law
      a  = 0.4    ;  %pole
      uk = uk1 + 0.1*ig*( (a-1)*yk + (1-a)*rk -FL*dx  );

      if uk>10,         %Upper limit for control
            uk=10;
      end;
      if uk<0,          %Lower limit for control
            uk=0;
      end;

      %************************************************************
      YR(:,time)=yk;
      UR(:,time)=uk;
      XN(:,time)=xn;
      ZN(:,time)=zn;
      YN(:,time)=yn;

      uk2=uk1; uk1=uk;
      yk2=yk1; yk1=yk;
      ek2=ek1; ek1=ek;
      vk2=vk1; vk1=vk;
      xn2=xn1; xn1=xn;
      yn2=yn1; yn1=yn;


end


plot([ YR' YN' RR' UR' ])
```

8.5. Matlab code for solving the Diophantine equation

```
function [S,R,T]=colpolos(B,A,Am,ord,tipo)

for i=length(Am)+1:length(A)            %makes Am to the right size
    Am=[Am 0];
end

if tipo==1                        %creates the Diofantine Equation
      M=zeros(2*ord+1);
      for i=1:ord
            M(i:i+ord, i)=A’;
      end
      for i=1:ord+1
            M(i:i+ord-1, i+ord)=B(2:ord+1)’;
      end
      M(2*ord+1,1:ord)=ones(1,ord);       %the sum of the regulator
                                          %parameters are ....
      BB=[ Am(2:ord+1)’-A(2:ord+1)’       %the right hand side vector
           zeros(ord,1)                   %is created
           -1 ];
      FG=inv(M)*BB;               %the equation is solved
      R=[1 FG(1:ord)’];           %and the parameters are extracted
      S=FG(ord+1:2*ord+1)’;
      if abs(sum(B))<0.001 %insurance that T is not getting too big
            T=10;
      else
            T=sum(Am)/sum(B);
      End

else                          %same thing without integral action
      M=zeros(2*ord-1);
      for i=1:ord-1
            M(i:i+ord, i)=A’;
      end
      for i=1:ord
            M(i:i+ord-1, i+ord-1)=B(2:ord+1)’;
      end

      BB=[ Am(2:ord+1)’-A(2:ord+1)’
           zeros(ord-1,1) ];

      FG=inv(M)*BB;

      R=[ 1 FG(1:ord-1)’   0 ];           % zero at end to have the
                                          %same size as integral
      S=[ FG(ord:2*ord-1)’ 0 ];
      if abs(sum(B))<0.001
            T=10;
      else
            T=sum(Am)/sum(B);
      end
end
```

References

Advantech Co., Ltd. (1994). *User's Manual, PCL-818L.*

Åström, Karl Johan and Wittenmark, Björn. (1997). *Computer-Controlled Systems: Theory and Design*, third edition. Prentice Hall.

Åström, Karl Johan and Wittenmark, Björn. (1995). *Adaptive Control*, second edition. Prentice Hall.

te Braake, H.A.B., van Can, H.J.L., Scherpen, J.M.A. and Verbruggen, H.B. (1997). *Control of Nonlinear Chemical Process Using Neural Models and Feedback Linearization*. Computers & Chemical Engeneering.

te Braake, H.A.B., Ayala Botto, M., van Can, H.J.L., Sá da Costa, J. and Verbruggen, H.B. (1999). *Linear Predictive Control Based on Approximate Input-output Feedback Linearisation.* IEE Proc.-Control Theory Appl., Vol.146, No.4.

Feedback. (1986). *Technical Information, Process Trainer PT 326.*

Henriques, J., Castillo, B., Dourado, A. and Titli, A. (Not yet published, but submitted to the IEEE international conference on control and instrumentation, IECON-2000). *A Recurrent Neuronal Approach for the Nonlinear Discrete Time Output Regulation.*

Isidori. (1995). *Nonlinear Control Systems.* Springer Verlag.

Johansson, Rolf. (1993). *System Modelling & Identification*. Prentice Hall.

Landau, I. D., Lozano, R., M'Saad, M. (1997). *Adaptive Control*. Springer Verlag.

Matlab Compiler 1.0. (1995). MathWorks, Inc.

Matlab Neural Network Toolbox 3.0.1. (1998). MathWorks, inc.

de Wilde, Philippe. (1997). *Neural Network Models*, second edition. Springer Verlag.

Wellstead, P.E. and Zarrop, M.B. (1991). *Self-Tuning Systems; Control and Signal Processing*. John Wiley & Sons.