Neural Network Control

Daniel Eggert

24th February 2003

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 4525 3351, Fax +45 4588 2673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-THESIS: ISSN 1601-233X

Abstract

This thesis addresses two neural network based control systems. The first is a neural network based predictive controller. System identification and controller design are discussed. The second is a direct neural network controller. Parameter choice and training methods are discussed. Both controllers are tested on two different plants. Problems regarding implementations are discussed.

First the neural network based predictive controller is introduced as an extension to the generalised predictive controller (GPC) to allow control of non-linear plant. The controller design includes the GPC parameters, but prediction is done explicitly by using a neural network model of the plant. System identification is discussed. Two control systems are constructed for two different plants: A coupled tank system and an inverse pendulum. This shows how implementation aspects such as plant excitation during system identification are handled. Limitations of the controller type are discussed and shown on the two implementations.

In the second part of this thesis, the direct neural network controller is discussed. An output feedback controller is constructed around a neural network. Controller parameters are determined using system simulations. The control system is applied as a single-step ahead controller to two different plants. One of them is a path-following problem in connection with a reversing trailer truck. This system illustrates an approach with step-wise increasing controller complexity to handle the unstable control object. The second plant is a coupled tank system.

Comparison is made with the first controller. Both controllers are shown to work. But for the neural network based predictive controller, construction of a neural network model of high accuracy is critical – especially when long prediction horizons are needed. This limits application to plants that

can be modelled to sufficient accuracy.

The direct neural network controller does not need a model. Instead the controller is trained on simulation runs of the plant. This requires careful selection of training scenarios, as these scenarios have impact on the performance of the controller.

Lyngby,
den 24. februar 2003

Daniel Eggert

Contents

Chapter 1
Introduction

The present thesis illustrates application of the feed-forward network to control systems with non-linear plants. This thesis focuses on two conceptually different approaches to applying neural networks to control systems.

Many areas of control systems exist, in which neural networks can be applied, but the scope of this thesis limits the focus to the following two approaches.

The first application uses the neural network for system identification. The resulting neural network plant model is then used in a predictive controller. This is discussed in chapter 2.

The other control system uses neural networks in a very different way. No plant model is created, but the neural network is used to directly calculate the control signal. This is discussed in chapter 3.

Both chapters discuss theoretic aspects and then try to apply the control system to two separate plants.

Is is important to note that this thesis is not self-contained. Many aspects are merely touched upon and others are not at all covered here. Instead this thesis tries to give an overall picture of the two approaches and some of their forces and pitfalls.

Due to the limited scope of this thesis, no attempt has been made to discuss the issues of noise in conjunction with the described control systems. We assume that the data sources are deterministic.

Likewise the stability of presented controllers will not be discussed either.

Finally, it is worth noting that the plants used throughout this thesis are merely mathematical models in form of ordinary differential equations

(ODEs). All runs of the system are exclusively done by simulation on a computer – even when this is not mentioned explicitly.

1.1 Overview

After a list of used symbols, this chapter will briefly summarise the kind of neural network used in this thesis.

Then chapter 2 and chapter 3 each discuss one controller type. All findings are then discussed in chapter 4.

The Matlab source files written for this thesis can be found in appendix A.

1.2 The Neural Network

Throughout this thesis we are using the two-layer feed-forward network with sigmoidal hidden units and linear output units.

This network structure is by far the most widely used. General concepts translate to other network topologies and structures, but we will limit our focus to this network and shortly summarise its main aspects.

1.2.1 The Two-Layer perceptron

The neural network used in this thesis has a structure as sketched in figure 1.1.

The d input units feed the network with signals, p_i. Each of the M hidden units receive all input signals – each of them multiplied by a weight. The summed input a_j of the jth hidden unit is calculated from the input signals as follows:

$$a_j = \sum_{i=0}^{d} w_{ji} p_i \tag{1.1}$$

where $p_i, 1 \leq i \leq d$ are the inputs and $p_0 := 1$, so that w_{j0} is the bias of the jth hidden unit. In this way we absorb the bias into the weights.

The activation of the jth hidden unit is $g(a_j)$ where $g(\cdot)$ is the hidden unit activation function.

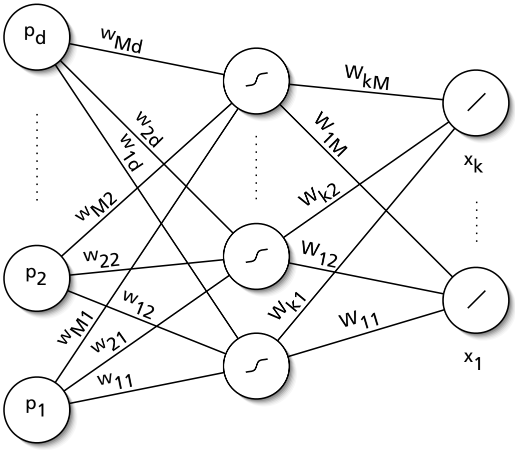


Figure 1.1: The two-layer perceptron with d inputs, M hidden units and k outputs.

In the same way the activation of the kth output unit is

$$x_k = \tilde{g}\left(\sum_{j=0}^{M} W_{kj} \cdot g(a_j)\right) \tag{1.2}$$

Also here we have absorbed the bias into the weights, by setting $g(a_0) := 1$ which results in W_{k0} being the bias for the kth output unit.

By inserting (1.1) into (1.2) we get

$$x_k = \tilde{g}\left(\sum_{j=0}^{M} W_{kj} \cdot g\left(\sum_{i=0}^{d} w_{ji} p_i\right)\right) \tag{1.3}$$

We will be using a sigmoidal activation function: the hyperbolic tangent $\tanh$ plotted in figure 1.2, for the hidden units:

$$g(a_j) \equiv \tanh(a_j) \equiv \frac{e^{a_j} - e^{-a_j}}{e^{a_j} + e^{-a_j}} \tag{1.4}$$

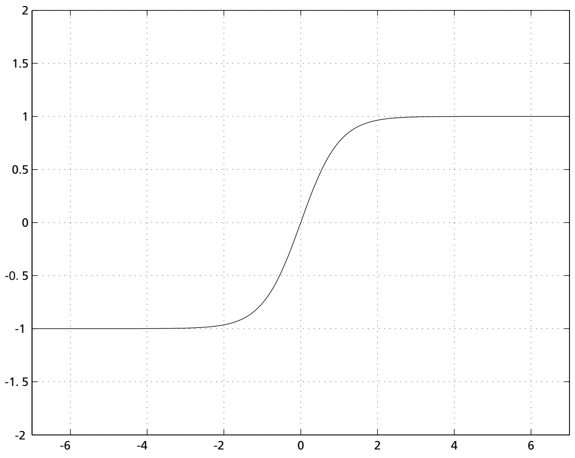


Figure 1.2: The hyperbolic tangent that is used as activation function for the hidden units.

and a linear activation function for the output units:

$$\tilde{g}(\tilde{a}_k) \equiv \tilde{a}_k$$

such that the kth output x_k is given by

$$x_k = \sum_{j=0}^{M} W_{kj} \cdot \tanh\left(\sum_{i=0}^{d} w_{ji} p_i\right) \tag{1.5}$$

The network with the functional mapping (1.5) maps a multi-variable input into a multi-variable output. Any functional continuous mapping can be approximated by this neural network to an arbitrary precision provided the number of hidden units M is sufficiently large.

1.2.2 Training

The process of tuning the neural network weights in order to achieve a certain kind of performance of the neural network is called training.

In general, some kind of error function E is specified, and the training algorithm will search for the weights that result in a minimum of the error function.

The two conceptually different approaches of applying a neural network to a control context in chapter 2 and 3 respectively, result in two very different error functions and hence two different training approaches. Each of these are discussed in the mentioned chapters.

Chapter 2

Neural Network Model Based Predictive Control

The aim of controller design is to construct a controller that generates control signals that in turn generate the desired plant output subject to given constraints.

Predictive control tries to predict, what would happen to the plant output for a given control signal. This way, we know in advance, what effect our control will have, and we use this knowledge to pick the best possible control signal.

What the best possible outcome is depends on the given plant and situation, but the general idea is the same.

An algorithm called *generalised predictive control* was introduced in [1] to implement predictive control, and we will summarise some of it below in section 2.1.

The algorithm is based on the assumption that the plant can be modelled with a linear model. If this is not the case, we can use a non-linear model of the plant to do the prediction of plant outputs. This chapter deals with using a neural network model of the plant to do predictive control.

First we discuss the linear case of predictive control. Then aspects of modelling of the plant with a neural network are investigated. This is also referred to as system identification. Section 2.3 looks into implementation of the algorithm for a coupled tank system, and section 2.4 discusses the implementation for an inverse pendulum, the so-called acrobot.

2.1 Generalised Predictive Control

The generalised predictive control (GPC) is discussed in [1] [2]. It is a receding horizon method. For a series of projected controls the plant output is predicted over a given number of samples, and the control strategy is based upon the projected control series and the predicted plant output.

We will shortly summarise some of the features of the GPC but refer to [1] [2] for a more detailed discussion.

The objective is to find a control time series that minimises the cost function

$$J_{GPC} = E\left\{ \sum_{j=1}^{N_2} (y(t+j) - r(t+j))^2 + \rho \cdot \sum_{j=1}^{N_2} (\Delta u(t+j-1))^2 \right\} \tag{2.1}$$

subject to the constraint that the projected controls are a function of available data.

N_2 is the costing horizon. ρ is the control weight. The expectation of (2.1) is conditioned on data up to time t.

The plant output is denoted y, the reference r, and the control signal u. Δ is the shifting operator $1 - q^{-1}$, such that

$$\Delta u(t) = u(t) - u(t-1)$$

The first summation in the cost function penalises deviation of the plant output $y(t)$ from the reference $r(t)$ in the time interval $t + 1 \leq t \leq t + N_2$. The second summation penalises control signal change $\Delta u(t)$ of the projected control series.

We will furthermore introduce the so-called control horizon $N_u \leq N_2$. Beyond this horizon the projected control increments Δu are fixed at zero:

$$\Delta u(t+j-1) = 0, \qquad j > N_u$$

The projected control signal will remain constant after this time. This is equivalent to placing an effectively infinite weight on control changes for $t > N_u$. Small values of N_u generally result in smooth and sluggish actuations, while larger values provide more active controls. The use of a control horizon $N_u < N_2$ reduces the computation burden. In the non-linear cast the reduction is dramatic. [1]

2.1.1 The Control Law for Linear Systems

Let us first derive the control law for a linear system, that can be modelled by the CARIMA model[1]

$$A(q^{-1})y(t) = q^{-1}B(q^{-1})u(t) + C(q^{-1})\xi(t)/\Delta \tag{2.2}$$

where $\xi(t)$ is an uncorrelated random sequence.

We introduce

$$\begin{aligned}
\mathbf{y}_N &= [y(t+1), y(t+2), \ldots, y(t+N_2)]^T && (2.3)\\
\mathbf{w} &= [r(t+1), r(t+2), \ldots, r(t+N_2)]^T && (2.4)\\
\tilde{\mathbf{u}} &= [\Delta u(t), \Delta u(1+2), \ldots, \Delta u(t+N_2-1)]^T && (2.5)
\end{aligned}$$

and let $\mathbf{y}_t$ represent data up to time t. We now note, that

$$\begin{aligned}
J &= E\left\{ \|\mathbf{y}_N - \mathbf{w}\|^2 + \rho\|\Delta\mathbf{u}\|^2 \middle| \mathbf{y}_t \right\} \\
&= \|\hat{\mathbf{y}}_N - \mathbf{w}\|^2 + \rho\|\tilde{\mathbf{u}}\|^2 + \sigma && (2.6)
\end{aligned}$$

where in turn

$$\hat{\mathbf{y}}_N = E\{\mathbf{y}_N | \tilde{\mathbf{u}}, \mathbf{y}_t\} \tag{2.7}$$

In linear case σ is independent of $\tilde{\mathbf{u}}$ and $\mathbf{y}_N$, and when minimising the cost function, σ constitutes a constant value that can be ignored. We can separate $\hat{\mathbf{y}}_N$ into two terms: [1]

$$\hat{\mathbf{y}}_N = \mathbf{G}\tilde{\mathbf{u}} + \mathbf{f} \tag{2.8}$$

where $\mathbf{f}$ is the *free response*, and $\mathbf{G}\tilde{\mathbf{u}}$ is the *forced response*.

The free response is the response of the plant output as a result of the current state of the plant with no change in input. The forced response is

[1]The Controlled Auto-Regressive and Moving-Average model is used in [1].

the plant output due to the control sequence $\tilde{\mathbf{u}}$. The linearity allows for this separation of the two responses.

The important fact is that $\mathbf{f}$ depends on the plant parameters and $\mathbf{y}_t$ (i.e. past values of u and y) while the matrix $\mathbf{G}$ only depends on the plant parameters. $\mathbf{G}$ does not change over time, and $\mathbf{f}$ can be computed very efficiently. [2,3]

Using (2.6) and (2.8) we now have

$$\begin{aligned} J &= \|\hat{\mathbf{y}}_N - \mathbf{w}\|^2 + \rho\|\tilde{u}\|^2 \\ &= \|\mathbf{G}\tilde{\mathbf{u}} + \mathbf{f} - \mathbf{w}\|^2 + \rho\|\tilde{u}\|^2 \\ &= (\mathbf{G}\tilde{\mathbf{u}} + \mathbf{f} - \mathbf{w})^T(\mathbf{G}\tilde{\mathbf{u}} + \mathbf{f} - \mathbf{w}) + \rho\tilde{\mathbf{u}}^T\tilde{\mathbf{u}} \end{aligned} \tag{2.9}$$

The minimisation of the cost function on future controls results in the following control increment vector: [1]

$$\min_{\tilde{\mathbf{u}}} J = \tilde{\mathbf{u}}^* = (\mathbf{G}^T\mathbf{G} + \rho\mathbf{I})^{-1}\mathbf{G}^T(\mathbf{w}\mathbf{f}) \tag{2.10}$$

and in the following current control signal:

$$u(t) = u(t-1) + \bar{\mathbf{g}}^T(\mathbf{w} - \mathbf{f}) \tag{2.11}$$

where $\bar{\mathbf{g}}^T$ is the first row of $(\mathbf{G}^T\mathbf{G} + \rho\mathbf{I})^{-1}\mathbf{G}^T$.

2.1.2 Non-linear Case

If the plant is non-linear, the cost function (2.1) / (2.6) remains the same, but implementation of the algorithm is done in a different way.[4]

Especially the prediction of the plant outputs $\hat{\mathbf{y}}_N$ based on the projected control series $\tilde{\mathbf{u}}$ can not be done in the same way.

For a non-linear plant the prediction of future output signals can not be found in a way similar to (2.8): The relation of the control signal series to

[2]Note that we let $\mathbf{y}_t$ denote all data up to time t.

[3]The matrix $\mathbf{G}$ and the vector $\mathbf{f}$ are defined in [1].

[4]Note that σ in equation (2.6) is not independent of $\tilde{\mathbf{u}}$ nor $\mathbf{y}_N$ for a non-linear plant. We will however assume that the effects of σ are small and can be ignored.

the plant output series is non-linear, i.e. the multi-variable function $\mathbf{f}$, that fulfils

$$\hat{\mathbf{y}}_N = \mathbf{f}(\mathbf{y}_t, \tilde{\mathbf{u}}) \tag{2.12}$$

is non-linear. Therefore we can not separate the free response from the forced response.[5]

To solve this problem we need to implement the prediction of $\mathbf{y}_N$, i.e. $\mathbf{f}$ in (2.12) in a different way. We need a non-linear predictor $\hat{\mathbf{y}}_N$ for future plant outputs.

This is where the neural network comes into play: We will facilitate the neural network to the workings of the function $\mathbf{f}$ in (2.12) to obtain the predictions $\hat{\mathbf{y}}_N$. We will train a neural network to do a time series prediction of plant outputs $\hat{\mathbf{y}}_N$ for a given control signal time series $\tilde{\mathbf{u}}$. This will let us evaluate the GPC cost function (2.1) for a non-linear plant.

Using a suitable optimisation algorithm on the projected control signal series with respect to the cost function, we find a control series, that minimises the cost function:

$$\min_{\tilde{\mathbf{u}}} J = \tilde{\mathbf{u}}^* \tag{2.13}$$

The complete control algorithm iterates through these three steps:

- choose (a new) control change time series $\tilde{\mathbf{u}}$
- predict plant output times series $\hat{\mathbf{y}}$
- calculate cost function J

That is, first we choose a control time series $\tilde{\mathbf{u}}$. Using a model we look at what would happen if we choose this series of control signals: we predict the plant output $\hat{\mathbf{y}}$. Then we use a cost function (2.1) to tell us how good or bad the outcome is. Then we let the optimisation algorithm choose a new control time series $\tilde{\mathbf{u}}$ that is better by means of the cost function (2.1). We iterate these steps in order to find a series of control signals that is optimal with respect to the cost function (2.1).

There are 3 distinctive components in this controller:

[5]See (2.3) and (2.5) for signal definitions.

- time series predictor of plant outputs
- cost function
- optimisation algorithm

The cost function is given by (2.1). For the optimisation algorithm we can use a simplex method which is implemented as a Matlab built-in function, such as `fminsearch`.

As we shall see below in section 2.1.3, the time series predictor will use a neural network model of the plant.

The complete system diagram is sketched in figure 2.1. The neural network model runs alongside the real plant and is used by the controller as discussed above to predict plant outputs to be used by the controller's optimisation algorithm.

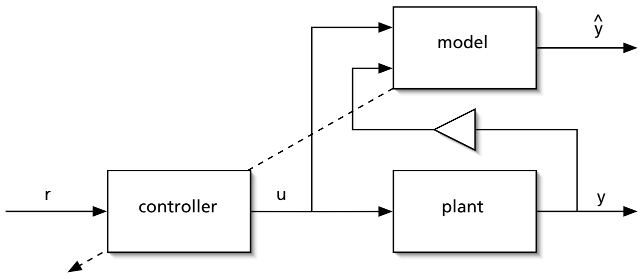


Figure 2.1: The neural network based predictive controller

Control Horizon

As noted earlier in section 2.1, the control horizon $N_u \leq N_2$ limits the number of projected control signal changes such that

$$\Delta u(t+j-1) = 0, \qquad j > N_u$$

If we set the control horizon to some value $N_u < N_2$, the optimiser will only work on a series of N_u projected control signal changes, while evaluating a predicted plant output series that has N_2 samples.

Reducing N_u will dramatically reduce the computational burden, as it effectively reduces the dimension of the multi-dimensional variable that the optimiser works on.

2.1.3 Time Series Prediction with Neural Networks

The purpose of our neural network model is to do time series prediction of the plant output. Given a series of control signals $\tilde{\mathbf{u}}$ and past data $\mathbf{y}_t$ we want to predict the plant output series $\mathbf{y}_N$ (Cf. equations (2.3)-(2.5)).

We will train the network to do *one-step-ahead* prediction, i.e. to predict the plant output y_{t+1} given the current control signal u_t and plant output y_t. The neural network will implement the function

$$\hat{y}_{t+1} = f(u_t, y_t) \tag{2.14}$$

As will be discussed below, y_t has to contain sufficient information for this prediction to be possible.[6]

To achieve *multi-step-ahead* prediction of all the plant outputs in (2.12) we will cycle the one step ahead prediction back to the model input. In this manner, we get predicted signals step by step for time $t+1, t+2, \ldots t+n$.[7]

One problem is that this method will cause a rapidly increasing divergence due to accumulation of errors. It therefore puts high demands on accuracy of the model. The better the model matches the actual plant the less significant the accumulated error.

A sampling time as large as possible is an effective method to reduce the error accumulation as it effectively reduces the number of steps needed for a given time horizon.

The neural network trained to do one-step-ahead prediction will model the plant. The acquisition of this model is also referred to as system identification.

[6]We assume that y_t is multi-variable.

[7]It is possible to train neural networks to take the control signal time series $(u_t, u_{t+1}, \ldots)$ as inputs and then output the complete resulting time series of plant outputs (2.12) in one go. It must be noted, though, that the demands on training data increase dramatically with increasing time series length n (curse of dimensionality). Furthermore, one might argue that if in fact training data of sufficient size was available this data could train a one step ahead network model to perform equally well.

Network inputs

In order to predict the plant output, the network needs sufficient information about the current state of the plant and the current control signal.

While this *state information* has to be sufficient to express the state of the plant, it is at the same time desirable to keep the number of states at a minimum: A larger number of states will increase the number of inputs and outputs of the neural network, which in turn will dramatically increase demands on the training data set size.

If we have a mathematical formula of the plant of the form

$$\dot{Y} = f(Y) \tag{2.15}$$

where Y is a vector of physical parameters, it is apparent that if we choose Y as our state vector, we will have sufficient information. If furthermore Y contains no redundant information, we must assume that no other states will describe the plant using fewer parameters.

Our network model can now be made to predict those states and we will feed the states back to the neural network model inputs. In this case, the neural network inputs are the (old) states Y and the control signal, u, as illustrated in figure 2.2.

For this to work we must be able to extract the state information from the plant in order to create our training data set. While this is not a problem when working with mathematical models in a computer, it will be a rare case in real life. The states in a description such as (2.15) can be difficult to measure.

A different approach is to use a lag network of control signals and plant outputs. If no state model (2.15) for the plant is known, this approach is feasible. The lag network has to be large enough for the network to extract sufficient information about the plant state.

We still want to keep the network inputs at a minimum since the demands on the training set size grow exponentially with the number of inputs. The size of the lag network, however, may have to be quite large in order to contain sufficient information.

We can effectively reduce the number of input nodes while still being able to use a sufficiently large lag network, by using principal component

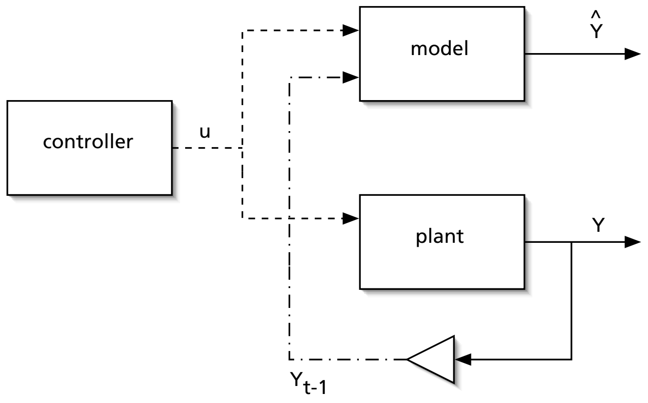


Figure 2.2: Feeding the plant states as an input to the plant model. The plant outputs its states Y, and those are delayed and fed into the neural network model of the plant, that will then be able to predict the plant output. During time series prediction, the output of the model itself will be fed back into its own input in order to do multi-step-ahead prediction as described in section 2.1.3.

analysis on the lag network outputs. Only a subset of the principal components is then fed into the neural network. This is discussed below as a part of pre-processing.

The input to the neural network would then consist of some linear combinations of past and present control signals and plant outputs, i.e. some linear combinations of

$$(u_{t-1}, u_{t-2}, \ldots, y_t, y_{t-1}, \ldots, r_t, \ldots). \tag{2.16}$$

The above concepts are unified in [3] by using a so called regressor. This regressor-function $\varphi(\cdot,\cdot)$ maps the known data (2.16) into a regressor φ_t of fixed dimension:

$$\varphi_t = \varphi(u_{t-1}, u_{t-2}, \ldots, y_t, y_{t-1}, \ldots, r_t, \ldots)$$

The regressor φ is then used as input to the neural network.[8]

2.2 System Identification

As noted above, we want the neural network to predict the plant states one time step ahead:

$$\hat{Y}_{t+1} = E\left\{Y_{t+1} \middle| Y_t, u_t\right\} \tag{2.17}$$

As noted above in section 7, we assume that Y_t contains sufficient information to describe the current state of the plant.

Or said in a different way, we want the neural network to implement the function f in

$$Y_{t+1} = f(Y_t, u_t) + \sigma_t \tag{2.18}$$

where σ_t is random noise.

To train the network, we use a data set that we obtain by simulation. During this simulation, the input signal needs to be chosen with care to assure that all frequencies *and* all amplitudes of interest are represented in this input signal as noted in [4].

[8]The regressor can also be used to incorporate some other aspects of pre-processing mentioned in section 2.2.3.

[4] suggests a *level-change-at-random-instances* signal. At random times the level is set to random values and kept constant between those times:

$$x_t = \begin{cases} x_{t-1} & \text{with probability } p \\ e_t & \text{with probability } 1-p \end{cases} \tag{2.19}$$

where e_t is a random variable.

This training data set contains the corresponding Y_{t+1} for each set (Y_t, u_t). (Y_t, u_t) is the input vector $\mathbf{p}$, and Y_{t+1} is the target vector $\mathbf{t}$. Our training data set can be written as $\{\mathbf{p}^n, \mathbf{t}^n\}$.

2.2.1 Error Function

We want the neural network to model the underlying generator of the training data rather than memorising the data. When the neural network is faced with new input data, we want it to produce the best possible prediction for the target vector $\mathbf{t}$.[9] [5]

The probability density $p(\mathbf{p}, \mathbf{t})$ gives the most complete and general description of the data. For this reason the likelihood of the training data – with respect to this density – is a good measure of how good the neural network models the underlying data generator.

The likelihood of the training data can be written as

$$\mathcal{L} = \prod_n p(\mathbf{p}^n, \mathbf{t}^n) = \prod_n p(\mathbf{t}^n|\mathbf{p}^n)p(\mathbf{p}^n) \tag{2.20}$$

The aim of the training algorithm is now to maximise this likelihood.

Since it is generally more convenient to minimise the negative logarithm of the likelihood, we introduce the *error function*

$$E = -\ln \mathcal{L}$$

If we assume that the distribution of the target data is Gaussian and furthermore remove additive constants from the cost function, we get the

[9]The problem of fitting the underlying data generator instead of memorising the data is discussed further in section 2.2.5 and 2.2.6 where the concept of regularisation is introduced.

following *sum-of-squares* cost function:[10]

$$\begin{aligned} E &= \frac{1}{2}\sum_{n=1}^{N}\sum_{k} = 1^{c}\,[x_k(\mathbf{p}^n;\mathbf{w}) - t_k^n]^2 \\ &= \frac{1}{2}\sum_{n=1}^{N} \|\mathbf{x}(\mathbf{p}^n,\mathbf{w}) - \mathbf{t}^n\|^2 \end{aligned} \tag{2.21}$$

The error function sums the squares of the deviation of the network outputs $\mathbf{x}^n$ for the given weights $\mathbf{w}$ from the target values $\mathbf{t}^n$. [5]

2.2.2 Error Back-propagation

The error function (2.21) tells us to what degree the neural network model fits the training data. We now need to consider how the network is trained to achieve a good model fit, i.e. how we can minimise the error function E.

This tuning of the neural network is done by adjusting the weight. The question is which weights are responsible for a poor performance. This is known as the *credit assignment problem.*

The solution is to evaluate the derivatives of the error E with respect to the neural network weights $\mathbf{w}$. The sum-of-squares error function (2.21) is a differentiable function. We use the derivatives to find the weights, that minimise the error function by using an optimisation method such as gradient decent. The algorithm we have used is the more powerful Levenberg-Marquardt algorithm. It generally results in faster training. This algorithm is discussed in detail in [5].

Let us shortly summarise the general concept of this optimisation algorithm.

The evaluation of the derivatives of the error function is called *error back-propagation* because the errors are propagated backwards through the network.

[10]For a more detailed derivation see [5]. Note that $\mathbf{w}$ are the network weights, x_k the kth network output, and t_k^n the target value for the kth output for the nth data vector of the training set.

The error function 2.21 is a sum over the errors for each pattern in the training data set in the following way

$$E = \sum_n E^n$$

The error function E^n for the pattern n can be written as a differentiable function of the network output variables: $E^n = E^n(x_1, \ldots, x_c)$. The derivatives of the error function E can now be expressed as the sum over the training set patterns of the derivatives for each pattern.

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E^n}{\partial w_{ji}} \tag{2.22}$$

Let us therefore look at the derivative of the error function E^n for one pattern. Using the notation of section 1.2.1 we introduce δ, a so called *error* for each unit as follows

$$\delta_j \equiv \frac{\partial E^n}{\partial a_j} \tag{2.23}$$

For each output unit k this results in

$$\begin{aligned} \delta_k &\equiv \frac{\partial E^n}{\partial \tilde{a}_k} = \tilde{g}'(a_k)\frac{\partial E^n}{\partial x_k} \\ &= y_k - t_k \end{aligned} \tag{2.24}$$

and for the hidden units this results in

$$\begin{aligned} \delta_j &\equiv \frac{\partial E^n}{\partial a_j} = \sum_k \frac{\partial E^n}{\partial \tilde{a}_k}\frac{\partial \tilde{a}_k}{\partial a_j} \\ &= g'(a_j)\sum_k W_{kj}\delta_k \end{aligned} \tag{2.25}$$

The derivatives with respect to the weights are now given by

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j p_i \qquad \frac{\partial E^n}{\partial W_{kj}} = \delta_k g(a_j) \tag{2.26}$$

The back-propagation procedure now works as follows: [5]

1. Apply the input $\mathbf{p}^n$ to the neural network input and find the activations of all units
2. Evaluate all δ_k for the output units using (2.24)
3. Back propagate these values using (2.24) to find all δ_j for the hidden units
4. Evaluate the derivatives with respect to the weights using (2.26)

As noted above, the derivatives are summed up over all training patterns as noted in (2.22).

Once we know what all derivatives are, we can update the weights. As noted above several strategies for parameter optimisation exist. The simplest is the fixed-step gradient descent technique. By summing up the derivatives over all the patterns in the training set, the weights are updated using

$$\Delta w_{ji} = -\eta \sum_n \delta_j^n x_i^n \tag{2.27}$$

where η is the step length.

We will be using the Levenberg-Marquardt algorithm that is described in textbooks such as [5]. It generally results in faster training.

2.2.3 Pre-processing and post-processing

If we have a-priori knowledge, there is no need to re-invent this knowledge with the neural network. If we instead move this knowledge *outside the neural network*, we will let the neural network focus on what we don't know, yet. Pre-processing and post-processing is one way of utilising a-priori knowledge by moving it outside the context of the neural network. Proper pre-processing and post-processing can effectively improve network performance.

A very simple, yet effective transformation of input and output data is to transform all inputs and outputs to zero mean and unit standard deviation. This way, all input and output levels will be of equal magnitude. The neural network model will not have to model the mean and standard deviation of signals, and this will result in better convergence of the neural network error function.

We can use the available training data set to estimate the mean and standard deviation of the neural network inputs and outputs. Then we use a linear transformation, that will map all signals to zero mean and unit standard deviation based on those estimates.

Another case where pre-processing can be of use, is if we know (or suspect) the input vector to contain redundant information. We want to keep the number of neural network inputs at a minimum without loosing valuable information contained in the input vector. Principal component analysis is a tool to reduce the dimension of the input vector while minimising information loss. We have not used principal component analysis in this present work.[11]

2.2.4 Model order selection

A network of too low order will not be able to fit the training data very well – it will not be flexible enough.

With growing model order the computation work needed for training will increase, and – what is worse – the demands on the size of the training data set grow exponentially. Simply put, n input and output variables span an n-dimensional space. The training data set has to *fill up* this space. This fact explains the requirement for an exponential growing data set and is referred to as the *curse of dimensionality*.

Finally we have to ensure that the neural network will not over-fit. We want the network to stay generic. If we present it with a new set of data, we expect it to perform equally well on this set, as it did on the training set.

The topic of formal model order selection is beyond the scope of this thesis. It is a complex topic, and many methods' complexity and demand for questionable assumptions make a *trial and error* approach a plausible alternative.

Furthermore it must be noted that if we choose a model order that is too high, but which we *can* train successfully, pruning methods such as *optimal*

[11]Principal components analysis is also know as Karhunen-Loéve transformation.

brain surgeon allow us to reduce the model order in a hands-on way, while keeping an eye on the network performance.[12]

Whichever method for model order selection is the most feasible depends very much on the problem at hand and available computation power. We shall not dig into this any further.

Let us turn towards methods for ensuring good generalisation, i.e. methods that will keep the neural network from over-fitting training data while allowing the network to be flexible.

2.2.5 Regularisation

As noted above, we need to avoid over-fitting on the training data. One way of doing this is to apply regularisation. In the following we shall shortly summarise the general concept.

Training of the neural network aims at minimising a given error function E that describes how well the neural network fits the training data.

If we use regularisation, we use a modified error function $\tilde{E}$ by adding to the standard error function E a penalty Ω, such that

$$\tilde{E} = E + \phi\Omega$$

where ϕ is a scalar weight on the penalty function. The penalty function Ω enforces a penalty on network weights, that over-fit the training data.

Different approaches for choosing the penalty function Ω exist. One of the simplest and most widely used regularisers is called weight decay, in which the penalty function consists of the sum of the squared weights. We shall not go any further into this topic as it is well covered in neural network literature such as [5]. We will use *early stopping* instead of regularisation as noted below in section 2.2.6.

2.2.6 Neural Network Training

During training we minimise the error function introduced in section 2.2.1. This way the neural network will be made to model the structure of the training data.

[12]Pruning algorithms are not discussed here. Refer to [5] for more on this topic.

We train the neural network using back propagation as discussed above in section 2.2.2. We're using the Levenberg-Marquardt algorithm for weight updating, and this algorithm is implemented in the Matlab *Neural Network Toolbox*.

To get a decent starting point for the network weights, we use a method similar to what is called the *hold out method:* We train 6 networks with random start guesses using the training data set. We only train 50 training epochs (steps). Then we evaluate the error function on the validation set and choose the network with the best performance. This network is then further trained using *early stopping*.

The initial cross-validation ensures, that our random start-guess is decent, and that our network is likely to have good convergence. We're less likely to be stuck in a local minimum far away from the optimal solution.

The neural network training is then used on this network until terminated by early stopping as explained below.

The overall training approach can be summarised as:

```
repeat 6 times
    choose random start weights for neural network
    train network for 50 epochs
end
choose best of above networks
train network using early stopping as termination
```

Early Stopping

It is important for the network to generalise well. The approach to ensure good generalisation we have chosen is called *early stopping*. We use two data sets for this method: One training data set and a separate validation data set.

While training the neural network using an error function on the training data set, we investigate the performance of the network on the validation data set. Poor generalisation will show as an increasing error function on the validation data set.

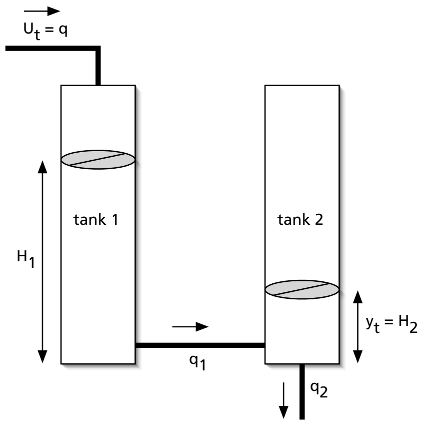


Figure 2.3: The coupled tank system has one input, the flow $u_t = q$ into tank 1, and one output, the height $y_t = H_2$ of tank 2.

Early stopping will stop the training algorithm once the error function increases on the validation set. In this way we avoid over-fitting.[13]

2.3 Implementation of the Coupled Tank System Controller

The first out of two plants to be controlled with the non-linear GPC is a coupled tank system. It is a single-input-single-output (SISO) system.

2.3.1 System Description

The system is illustrated in figure 2.3. The flow into tank 1 is the control signal u, while the level in tank 2 is the plant output y to be controlled.

[13]For quadratic error functions, early stopping gives rise to a behaviour similar to the one when using weight decay regularisation.[5], p. 345.

Physical sizes are:

$$\begin{array}{rclcrcl} C & = & 1.539m & & g & = & 9.81\frac{m}{s^2} \\ a_l & = & 5mm & & a_o & = & 8mm \\ \sigma_l & = & 0.44 & & \sigma_o & = & 0.31 \end{array}$$

Where C is the cross section area of the tanks, a_l and a_o are the cross sections of the pipe connecting the two tanks and the pipe leaving tank 2 respectively. σ_l and σ_o are the flow coefficients such that the flow between the tanks q_l and the flow out of tank 2 q_o are

$$\begin{array}{rcl} q_l & = & \sigma_l a_l \sqrt{2gY_2} \\ q_o & = & \sigma_o a_o \sqrt{2gY_1} \end{array} \tag{2.28}$$

where $Y_1 = H_2$ is the level in the second tank, and $Y_2 = H_1 - H_2$ is the difference in tank levels. The tank height is $0.6m$.[14]

Introducing

$$a_1 = \sigma_o a_o \frac{\sqrt{2g}}{C} \qquad a_2 = \sigma_l a_l \frac{\sqrt{2g}}{C}$$

we can write the plant's system equation as

$$\begin{array}{rcl} \dot{Y} & = & A \cdot Y + B \cdot u \\ \begin{bmatrix} \dot{Y}_1 \\ \dot{Y}_2 \end{bmatrix} & = & \begin{bmatrix} -a_1 & a_2 \\ a_1 & -2a_2 \end{bmatrix} \begin{bmatrix} \sqrt{Y_1} \\ \sqrt{Y_2} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{C} \end{bmatrix} u \end{array} \tag{2.29}$$

where u is the control signal. Note that for the plant output we have $y = Y_1$.

We will be using a sampling frequency of $T_s = 5s$ as suggested for this system by [6].

2.3.2 System Identification

In order to control the coupled tank system with a GPC we need to train a neural network model. General aspects are explained in sections 7 trough 2.2.6.

[14] We assume that $Y_1 \geq 0$ and $Y_2 \geq 0$. For the ODE solver to work properly at Y_1 or Y_2 close to zero we have to expand equation (2.28) to hold for negative values as well.

Model Structure

As we assume that the plant obeys (2.29), we will feed the states

$$Y = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix} = \begin{bmatrix} H_2 \\ H_1 - H_2 \end{bmatrix} \tag{2.30}$$

into the model as inputs along with the current control signal and train the network to predict the states for the next time step $(t + 1)$. Figure 2.2 on page 15 illustrates the interconnection.[15]

Altogether the neural network has three input nodes: The control signal and the tank levels at time t. There are two model outputs: the predicted tank levels for both tank 1 and tank 2 one time step ahead, i.e. at time $t + 1$.

Since we use early stopping, we're not troubled with over-fitting. This allows us to select a flexible network structure. We have chosen to use 12 hidden sigmoidal units. As we will see this number is large enough for the neural network to be able to model the plant. See also section 2.2.4.

The resulting model structure is sketched in figure 2.4.

Input Signal Design

In order to be able to train the neural network model (system identification), we need data that describes the entire operating range of the plant. Demands on input signals are stronger than for linear models. As stated in literature such as [4], the input signal should represent all amplitudes and frequencies.

Using a *level-change-at-random-instances* signal is suggested as it will give good excitation of the plant. This is true in a standard case. However, with the coupled tank system we can not use such a signal as input. This is due to the nature of the plant. If we did, the first tank would either flood or be at a low level all the time.

To get an excitation signal that will make the plant go through all its operating range, we need to investigate the coupled tank system more closely.

[15]Section 2.2.4 discusses the feasibility of this approach. The availability of state information is questionable in real-world implementation, but we will ignore this fact.

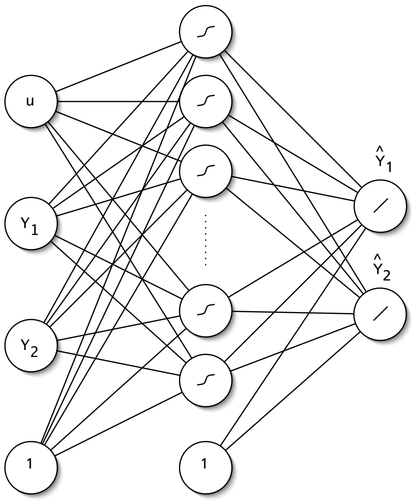


Figure 2.4: The structure of the neural network that is used to model the coupled tank system has 3 input units, 12 hidden sigmoidal units and 2 output units. Additionally there is one bias unit for both the input layer and the hidden layer (only 5 hidden units are shown here).

We need to find an appropriate *control strategy* that is both random and controlled. It needs to be random in order to excite the system and yield a rich data set for training, but at the same time it needs to be controlled in order to make the state of the tank to change through the entire operating range.

If we wanted to raise the level in tank 2, we would fill a lot of water into tank 1, and then stop filling in water. If we wanted to lower the level in tank 2, we would stop filling in water into tank 1 for some time. Our input signal needs to mimic this behaviour in some sort of way. At the same time, we need to note that the pump controlling the inflow into tank 1 can not operate in reverse. That is, we can not use negative control signals. While the model needs to learn this non-linearity near $u = 0$, there's no point in the control signal being negative for long periods of time.

We have all these observations put together into the following algorithm used for creating the input data:[16]

- As a starting point, we use a *level-change-at-random-instances* algorithm. The probability of changing the control signal for this algorithm is set to 5%. To help the control signal from wandering off into negative (and meaningless) values, there is a slight tendency for the change of the control signal to be positive.

- If the tank level in tank 2 is above a high mark of $0.55\mathrm{m}$ for more than 25 consecutive sampling periods, the control signal is fixed to zero for a random count of sampling periods.[17]

- If the control signal *wanders off* below zero for more than 60 sampling periods, the control signal is reset to some random, positive value.

This algorithm has proved to create data sets with rich excitation. One resulting training set with 40.000 samples is depicted in figure 2.5. The bottom graph shows the state of the algorithm.

[16]Implemented in `createtrainset.m` and `nnmodel.m` in section A.1.7 and A.1.6.
[17]The tank height is $0.6\mathrm{m}$ and units are SI.

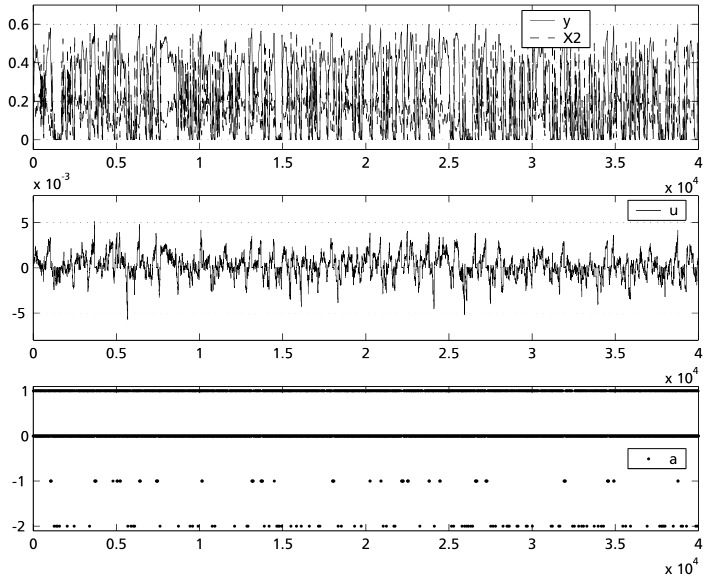


Figure 2.5: The training set with its 40,000 samples. $y = Y_1$ and Y_2 are the model *states* corresponding to the tank level in tank 2 (y) and the difference in tank levels (Y_2). The second graph shows the control signal, and the third graph shows the action of the input signal generation algorithm. $a = 0$ and $a = 1$ correspond to the *level-change-at-random-instances* algorithm, while $a = -1$ corresponds to a control signal fixed at zero (too high tank level), and $a = -2$ corresponds to a reset of the control signal because it has wandered of into negative values.

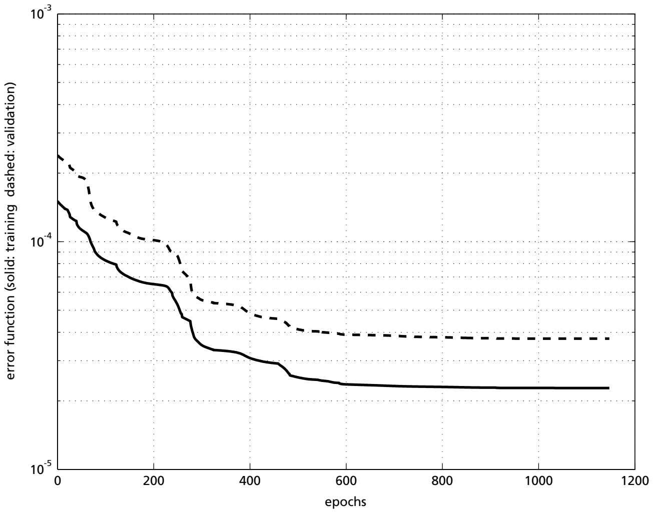


Figure 2.6: The neural network performance during training. The x-axis shows the training epochs / iterations. The performance is evaluated on both the training data set and the validation data set.

Once the data set is created, it is used to estimate the mean and variance of the neural network model input and output signals. The estimates are then used for pre-processing and post-processing as noted in section 2.2.3.

Neural Network Training

With the training data set and validation data set ready, the network is trained as explained in section 2.3.2. The performance during training is shown in figure 2.6.

Implementing Predictive Control

Using the neural network model with predictive control is done as noted in section 2.1.2. But the coupled tank system again needs special attention.

What is problematic in this case is the fact that a negative control signal doesn't have any effect beyond that of a zero control signal. The gradient for a negative control point hence is zero, and this is problematic for the off-the-shelf Matlab optimisation algorithm `fminsearch`.

What we need to do is to find the minimum of a constrained multi-variable function. The Matlab function `fminsearch` can minimise a multi-variable function, but it does not enforce constrains on the variables. In order to achieve this, we use iterative calls of this optimiser. Between calls we *adjust* the resulting variables (i.e. control signal) by re-setting negative values to zero:

```
repeat 8 times
    minimise U according to cost function
        using 4 * N_U iterations
    reset negative values of U to 0
end
```

The performance is discussed in the next section.

2.3.3 Performance

We will first look at the neural network model's ability to predict future plant outputs.

Time Series Prediction

To investigate the model's performance, we will feed the same random control signal into both the plant and the model. The random control signal is given by

$$u_t = \begin{cases} u_{t-1} + e_t & \text{with a probability of } 95\% \\ 1 \cdot 10^{-3} & \text{otherwise} \end{cases} \tag{2.31}$$

where e_t is a white noise sequence with zero mean and a variance of $\sigma_e^2 = 10^{-4}$.

A simulation series of 1000 samples and their corresponding one-step-ahead prediction are gathered. The resulting information is used to calculate the squared *multiple correlation coefficient* R^2:

$$R^2 = \frac{V(y) - V(\varepsilon)}{V(y)}$$

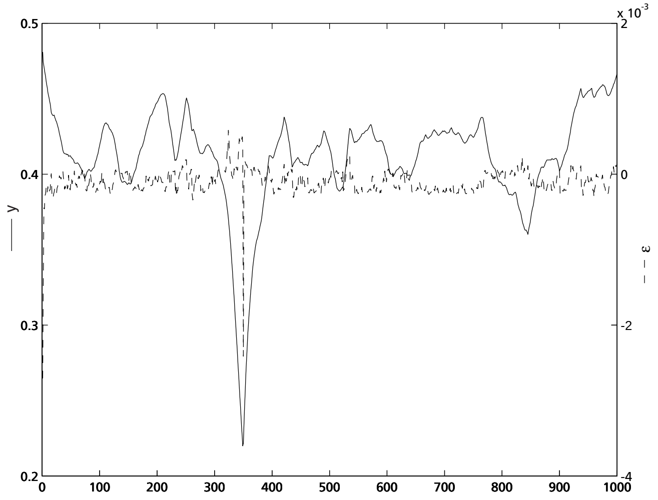


Figure 2.7: The model error ε compared to the plant output y for a prediction series. Note that the model error uses the right side axis, while the plant output uses the left side axis. The difference in magnitude is ≈ 200.

where $V(\cdot)$ is the variance, and $\varepsilon = y - \hat{y}$. The estimated variances result in

$$R^2 = 0.9935$$

This coefficient measures how much of the information on y is contained in the estimate $\hat{y}$. $R^2 = 0.9935$ is a high value and indicates good estimation.

Another way to investigate how good predictions are is to look at the cross-correlation coefficients.

Tests show that the error ε is correlated with both the plant output y and with the control signal u. This must be seen in light of the high multiple correlation coefficient R^2:

The error is very small compared to y and u as can be seen in figure 2.7 and hence the cross-correlation can not be used to support a higher model order.

Finally we will feed a random control signal as given by (2.31) into both

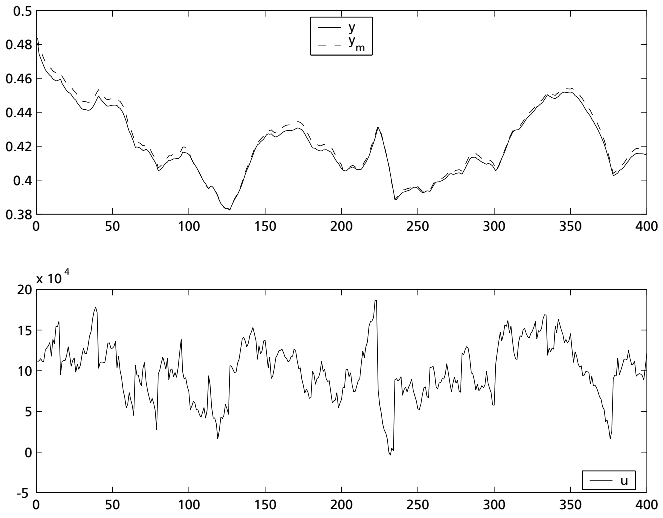


Figure 2.8: To test the model's ability to predict the plant output the same random control data is fed into the plant and the model. At every 100th sampling step the model is re-calibrated with the real plant. The resulting output is shown above. It is apparent that the model copes quite well with the prediction horizon of 100 steps.

the plant and the model. This time the model is doing a multi-step ahead prediction.

Figure 2.8 shows a simulation where the model is synchronised with the real plant at every 100 simulation steps. It is apparent that the neural network model can predict the plant output to a great degree of precision over a 100 samples horizon.

When selecting the horizon N_2 of the predictive controller it is important that the model can make accurate predictions within this horizon. Otherwise the iterative optimisation of the control strategy with respect to the cost function will not converge to any meaningful result.

With the above results values as high as $N_2 < 100$ seems reasonable, but we will use a smaller value as noted below.

Predictive Control

The algorithm for the predictive controller is discussed in section 2.1 and the following sections.

There are a number of parameters we have to choose: the control horizon N_u, the costing horizon N_2 and the control weight ρ.

Let us first look at the horizons. We have chosen the values [18]

$$N_u = 10 \qquad N_2 = 40$$

The accumulation of errors in the prediction of future plant outputs limits the horizon N_2. We can not increase N_2 beyond a certain limit, where the accumulated error starts to have impact.

As noted in [1], the costing horizon has to be large enough to extend over the dynamics of the plant. In other terms, the costing horizon has to be large enough to allow the algorithm to see the outcome of the predicted controls.

At the same time, an increased N_2 increases the computation time needed in a quasi-linear way, as we need to predict the plant output for a longer time series for each evaluation of the optimiser.

As we have seen above $N_2 = 40$ is within the model's ability to predict future outputs and it also covers the dynamics of the plant as we shall see in simulations done below.[19]

The choice of control horizon N_u has serious impact on the computational burden. As noted in section 2.1.2 increasing N_u will increase the dimension of the space in which the optimiser searches for an optimum. The limited computer power and the computation-overhead required by Matlab enforce the moderate choice of $N_u = 10$.

[18]Whether $N_2 = 40, N_u = 10$ are optimal is not evaluated here. The values are a trade off between computational burden and controller performance. They have been chosen most of all since they work well, not because they have been found to be optimal. [1] and [2] discus the choice of N_u and N_2 in more detail.

[19]Cf. figure 2.8 on page 33.

The third parameter to fix is ρ. We have chosen a value of $\rho = 100$, which allows rather large control signals.

Even with the above choice of parameters the computational burden was heavy. A simulation of the system with controller running 800 time steps, i.e. $800T_s = 4000s \approx 1h6'$ as shown in figures 2.9 and 2.10 took 2 hours of computation time.[20]

For this reason, the optimiser that searches for a projected control series is terminated after 320 iteration steps.[21] To help the divergence of the algorithm, the search for the optimal control sequence takes the sequence found at $t - 1$ shifted one time step and re-uses it as start guess for the projected control signal series.

Simulation-runs with these values are shown in figure 2.9. Two features are immediately evident: The noisy appearance of the control signal u and the dive of the plant output y preluding each increase in the reference signal r.

The noise-like appearance of the control signal u is an artefact from the termination of the optimiser on the projected control series. Increasing the number of iterations assigned to each optimisation run would help this situation, although real impact would be reached by decreasing the control horizon N_u.

The dive of the plant output y preluding an increase in the reference signal r may look strange at first, but it is a result of the predictive nature of the controller. By letting the plant output y go a bit below the reference signal r, the subsequent increase in plant output can be *shaped* in such a way, that minimises the control cost function (2.1).

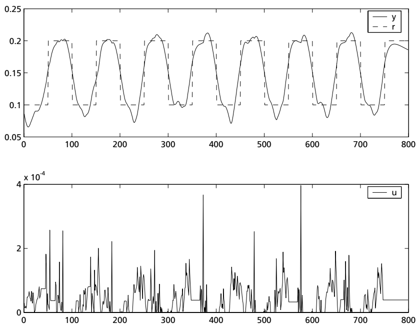
There are two more things that are important to note. The first one is best illustrated in figure 2.9(d), the results of a simulation with a level-change-at-random-instances reference signal. During the time interval

$$120 \leq t \leq 400$$

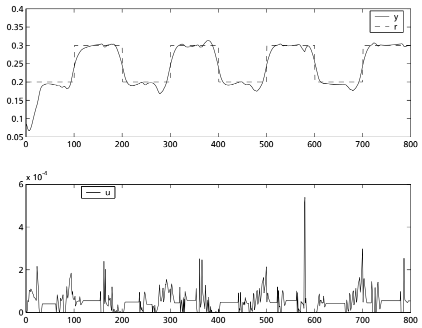
the reference signal is constant. The plant output is close to the reference,

[20]This time needed will decrease as computer power increases. Implementing the algorithm in a more efficient computer language would also dramatically improve performance, but this is beyond the scope of this thesis.

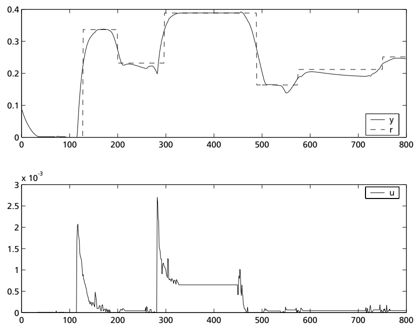
[21]The number of evaluations (i.e. predictions) done is somewhat bigger.



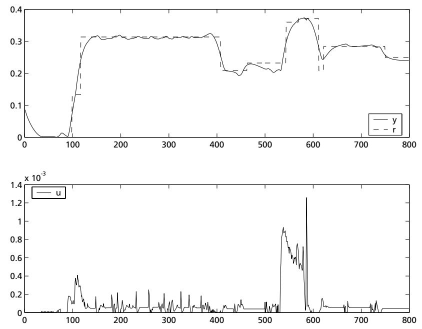
(a) reference is square wave with period of $100T_s$



(b) reference is square wave with period of $200T_s$



(c) reference is a level-change-at-random-instances signal



(d) reference is a level-change-at-random-instances constant signal

Figure 2.9: Simulation-runs of the neural network model based general predictive controller. Note that the axes of the control signal u have a different scale for figures 2.9(c) and 2.9(d) as compared to figure 2.9(a) and 2.9(b). This explains – to some extend – why the control signal appears to flutter more in the first two graphs.

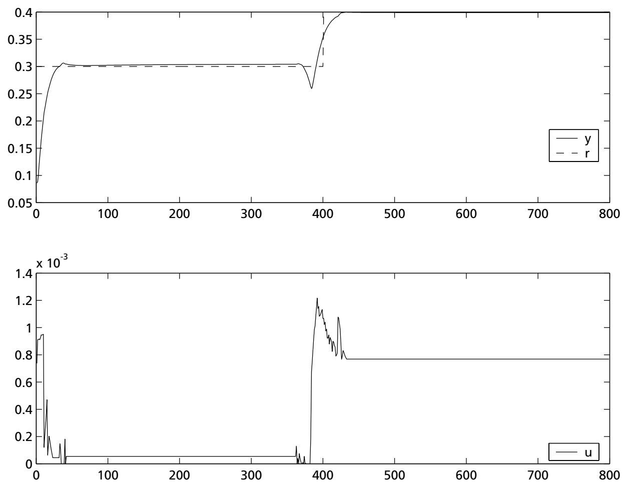


Figure 2.10: Simulation-runs of the neural network model based general predictive controller with a step reference signal

but it ripples about the reference. The reason for this is the behaviour of the control signal which in turn is due to the termination of the optimiser as noted above.

The other one concerns the behaviour shown in figure 2.9(c). Here the plant output drifts away from the reference in the time interval $600 \leq t \leq 750$. This is not the diving behaviour discussed above. Note that the costing horizon is only $N_2 = 40$.

The explanation is more cumbersome and includes some of the above. Note that we at this point need to hold the plant output at a level of $r \approx 0.2$ for a long period of time. This requires the input signal to be small for that period of time.

The problem with small input signals is that there is a discontinuity at $u = 0$. The control signal u can not go below $u = 0$, or said in a different way, the gradient of u on the negative side of $u = 0$ is non-existent or zero (depending on interpretation). This results in very poor performance of the simplex optimiser for values of u close to $u = 0$.

For the same reason, the control is more *calm* for higher values, such as in figure 2.9(c) for $350 \leq t \leq 450$ and in figure 2.10 for $t \geq 450$.

2.3.4 Discussion

Overall the coupled tank system shows that the approach to neural network based predictive control illustrated in this test is *doable* – at least for this system.

But at the same time it emphasises the fact that this straight forward concept results in many problems that are not easily overcome.

The predictive control algorithm needs a good predictor and this problem was solved. Demands for predicting over a time horizon that incorporates the plant's dynamics were met.

The problem of optimising the projected control signals was not solved in a satisfactory way. The resulting control signal is very sluggish and to some extend noisy due to convergence problems of the optimiser. The hard constraint of the control signal $u > 0$ caused problems with the simplex optimiser.

Never the less, the controller is able to control the plant. Even though the noisy nature of the control signal propagates to the plant output, the plant output follows the reference as one would expect for a receding horizon method.

Additional work on a better optimisation algorithm for the projected control series is likely to move the performance of the controller to different levels.

2.4 Implementation of the Acrobot Controller

The second of the two plants faced with the non-linear GPC is the so-called acrobot as implemented in [7]. The acrobot is a highly non-linear inverse pendulum.

We want the control system to keep both arms pointing upwards. A linear and a pseudo linear controller have been implemented in [7], both with a very limited region of attraction due to the systems non-linearity.

As we shall point out, general predictive control of this system faces severe difficulties and no working implementation was found. Simply put, this is

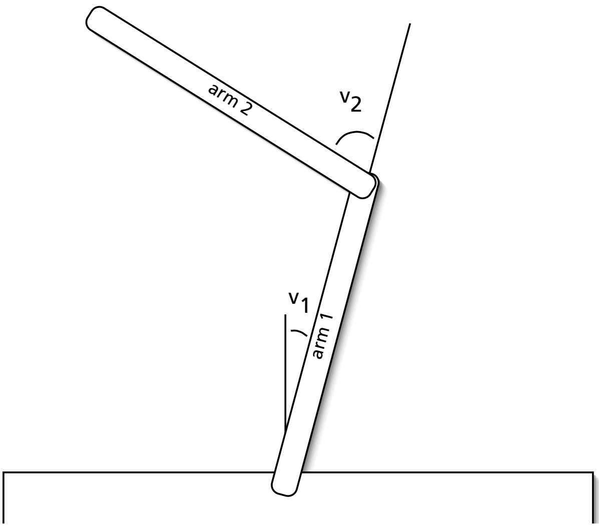


Figure 2.11: The acrobot consists of two arms. The end of the first arm is mounted on a bar and can freely rotate round this end. The other end connects to the second arm. A servo can act on this joint and create a moment on the second arm.

due to problems with modelling the acrobot and the length of the prediction horizon required.

2.4.1 System Description

The acrobot is sketched in figure 2.11. Although several equilibriums exist for the angles ν_1 and ν_2, we are interested in controlling the system such that the angles stay close to $(\nu_1, \nu_2) = (0, 0)$ which corresponds to an upright position of both arms.

Derivation of the kinematic equations is done with some detail in [7]. The results are, that the kinematics of the acrobot are governed by the following ordinary differential equation (ODE):

$$\dot{y} = \begin{bmatrix} y_3 \\ y_4 \\ \frac{-a_2(y_4^2-2*y_3*y_4)\sin(y_2)+a_4\sin(y_1)+a_5\sin(y_1-y_2)+a_3+a_2\cos(y_2)u}{a_1+2a_2\cos(y_2)} \\ u \end{bmatrix} \tag{2.32}$$

The state vector y is defined as

$$y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ \frac{dv_1}{dt} \\ \frac{dv_2}{dt} \end{bmatrix} \tag{2.33}$$

and the physical parameters are transformed into a_1, a_2 etc. as follows

$$A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} m_1 l_{c1}^2 + m2(l_1^2 + l_{c2}^2) + I_1 + I_2 \\ m_2 l_1 l_{c2} \\ m_2 l_{c2}^2 + I_2 \\ g(m_1 l_{c1} + m_2 l_1) \\ g m_2 l_{c2} \end{bmatrix}$$

The physical parameters values are

$$\begin{aligned} l_1 &= 0.438\text{m} & l_2 &= 0.410\text{m} \\ l_{c1} &= 0.102\text{m} & l_{c2} &= 0.100\text{m} \\ m_1 &= 3.49\text{kg} & m_2 &= 1.03\text{kg} \\ I_1 &= 91.3 \cdot 10^{-3}\text{kg m}^2 & I_2 &= 17.5 \cdot 10^{-3}\text{kg m}^2 \\ g &= 9.83\tfrac{\text{m}}{\text{s}^2} \end{aligned}$$

Notes on the Control Goal

As noted above, we want the acrobot to keep close to the upright $(v_1, v_2) = (0, 0)$ position. We will have to narrow down our design goal further as noted in the following.

The controllers constructed in [7] have a very limited region of attraction (ROA). The region of attraction for the linear controller suggested in [7] is limited to $\pm 1.5^\circ$ for the angle ν_2 while limits for the angle ν_1 are slightly wider. A second, so-called *Isidori* controller has a ROA of approximately $\pm 30^\circ$ for ν_1 and no limits on the second angle ν_2.

Adding to this the fact that both controllers take $2 - 4$s to *catch* the acrobot from within the controllers ROA, we need to limit our expectations as to what the model based controller will be able to do.

We will therefore target a ROA of size comparable to the one of the Isidori controller.

2.4.2 System Identification

In the following, we will summarise the system identification done for modelling the acrobot system with a neural network model.

We will use a neural network with a structure very much like the one used for the coupled tank system as described in section 2.3.2. The four parameters $y = [y_1, y_2, y_3, y_4]^T$ corresponding to (2.33) will be used as model states. The physical acrobot implemented in [7] allows measurement of all these states.

We will use y as the input to the neural network along with the current control signal u, while the network output will be the predicted state $\hat{y}$. This results in a total of 5 input signals and 4 output signals.

We have used as much as 30 hidden units, to get a very flexible network. The resulting model structure is sketched in figure 2.12.[22]

We note that the network has to model the wrap-around of the angles ν_1 and ν_2, i.e. the fact that the angle $2\pi - \delta$ is next to 0 (δ being some small value). It is furthermore important to note that the kinematic equations are quite complex in nature.

These two facts cause the desired input-output mapping to be highly non-linear and support the fact that we need to use a high number of hidden units.

[7] suggests a sampling frequency of $T_s = \frac{1}{64}$s. We have chosen to use the slightly faster $T_s = \frac{1}{100}$s for reasons of convenience (more straightforward interpretation of sample count vs. time).

[22]We would still need higher precision of the model in order to get a usable model based predictive controller. This is discussed later in section 2.4.3.

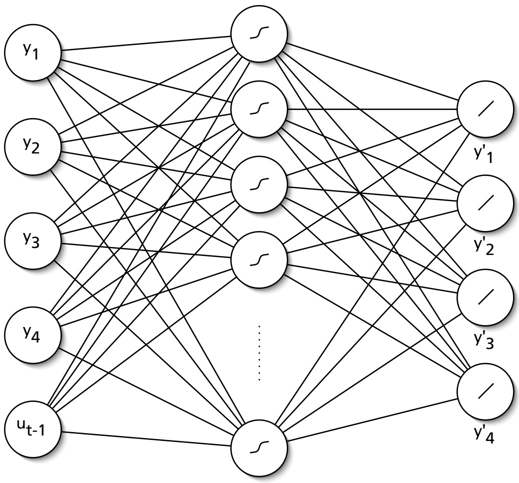


Figure 2.12: The structure of the neural network used to model the acrobot. Inputs are the current state along with the current control signal. Output is the next state. While only five are shown here, there are a total of 30 hidden sigmoidal units in the model.

Input Signal Design

The unstable nature of the acrobot makes it rather difficult to gather useful data for training the neural network model. We want to achieve a good model for the acrobot near the upright position. It is also important to note that the linear controller designed in [7] moves the angle ν_2 as low as $-60^\circ \simeq 1.0\mathrm{rad}$ to catch the arm from a starting point where $\nu_1 = \nu_2 = 1.2^\circ \simeq 0.0209\mathrm{rad}$. This tells us that we need a wide range of available data, especially for ν_2. (See figure 5, p. 22, [7])

Early attempts to start the acrobot near equilibrium and then use a linear controller to hold the acrobot near this position failed as the acrobot would *fall off* and quickly leave its region of attraction. Furthermore, the simple behaviour of the linear controller would limit the effective dimensionality of the training data set.

What turned out to be a more effective approach is to start the acrobot at a random state near the equilibrium situation and then use a combination of the earlier linear controller and a level-change-at-random-instances signal, i.e. there's a 20% chance, that we use the linear controller (as long as it generates a control signal within certain bounds) – otherwise we use a level-change-at-random-instances signal as control signal (Cf. section 2.2).

The control signal is hence given by

$$u_t = \begin{cases} -K \cdot Y_t & p_t > 0.8 \wedge |K \cdot Y_t| < 300 \\ \delta_t & \text{otherwise} \end{cases} \tag{2.34}$$

where $u_t = -K \cdot Y_{t-1}$ is the linear controller, p_t is a uniformly distributed random number sequence on the interval $[0,1]$, and δ_t is a level-change-at-random-instances signal.

This combination of the deterministic and control gives good excitation, but due to the unstable nature of the plant, the acrobot will soon leave the region that is of interest to us. We solve this problem by concatenating several short simulations as follows:

Once the acrobot leaves a defined range $\nu_1 < 2 \simeq 115^\circ$ we restart the simulation with a new starting point near equilibrium. In this way we gather a total of 1228 simulation series making up $69,980$ samples for the training data set. Figure 2.13 shows a few of these simulation series.

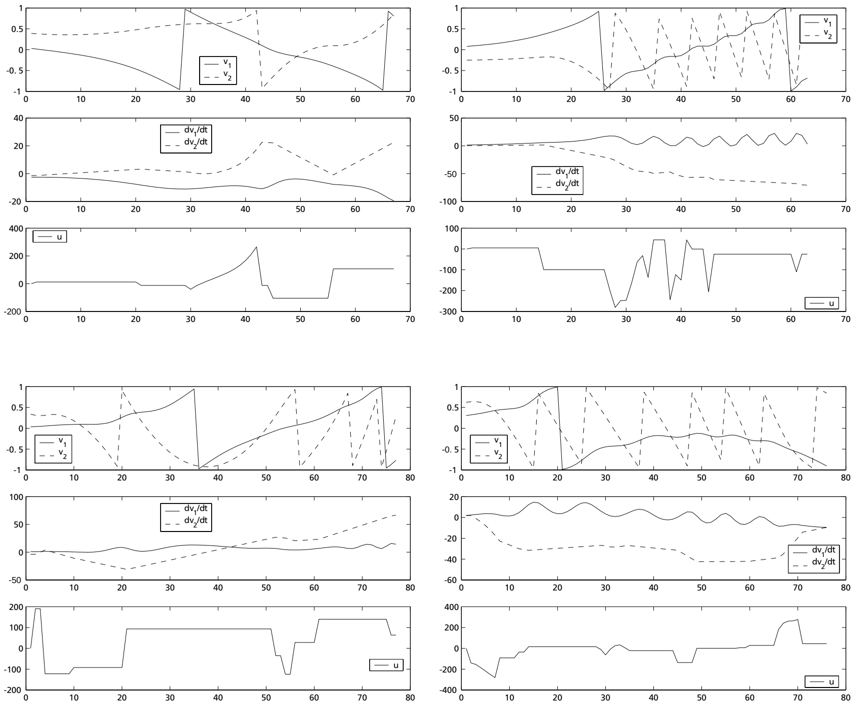


Figure 2.13: Short simulation sets such as the ones shown here are concatenated to form data for the training and validation data set.

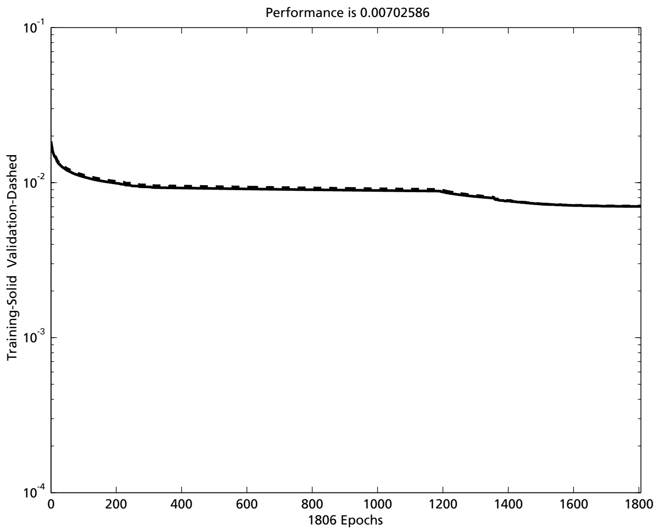


Figure 2.14: The neural network performance during training a model for the acrobot kinematics. The x-axis shows the training epochs / iterations. The performance is evaluated on both the training data set and the validation data set.

Before training the network, we have to do some preprocessing on the data. It is important only to use the modulus of 2π of the angles ν_1 and ν_2. The acrobot kinematics don't depend on how many times the angles have turned – only on the position relative to 0. Failing to do this would severely degrade the information available in the simulation results.

We furthermore use pre-processing and post-processing as noted in section 2.2.3 to get zero mean and unit standard deviation signals for inputs and outputs.

Neural Network Training

With the training and validation data set obtained as discussed above, training is done as described in section 2.2.6. The performance characteristics during training are shown in figure 2.14.

As we will see below, the training of the neural network fails by means of achieving an accurate model.

2.4.3 Performance

Let us first look at some simulations of time series predictions.

Time Series Prediction

In the same way as we did with the coupled tank system, we will initialise both the plant and the model of the plant in the same state, and then investigate the resulting time series of the two.

Figure 2.15 shows a few examples of simulation runs. The plant and the model are set in the same state and then the same control signals are fed into both the model and the plant. It is immediately apparent that this doesn't work very well.

While the predictions in figure 2.15(b) follow each other rather closely for all of the 25 simulation steps, all other plots in figure 2.15 show a sad story. The angles of the model hardly follow the plant's angles at all.

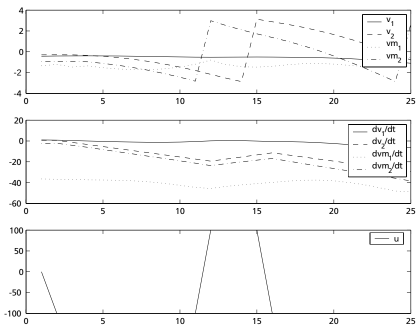
While the difference in angular velocity of angle ν_1 is at times huge, even the other offsets are by far too large in order to be useful for cost function evaluation.

Predictive Control

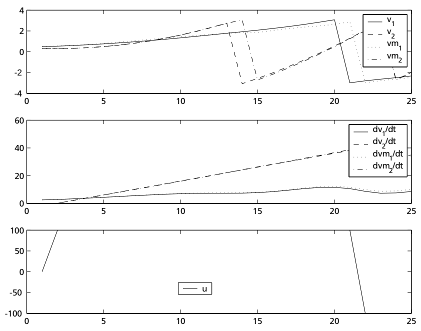
Since we can not do short term prediction of the plant output accurately, there is no hope that we can use a predictive controller on the acrobot system.

It is important to remember that other controllers, such as the Isidori controller suggested in [7] take almost $4s$ to stabilise the system from a start position where $\nu_1 = \nu_2 = 10^\circ \simeq 0.175 rad$.

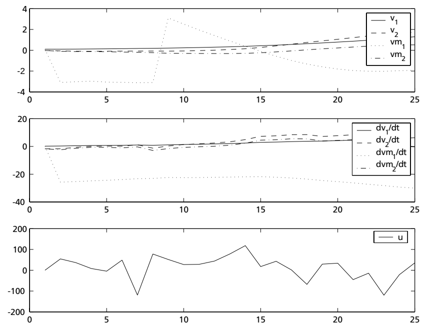
If the model based predictive controller at hand would have a similar performance, it would have to have a costing horizon N_2 that would cover approximately $4s$. With a sampling period of $T_s = \frac{1}{100}$ this equals to $N_2 = 4000$.



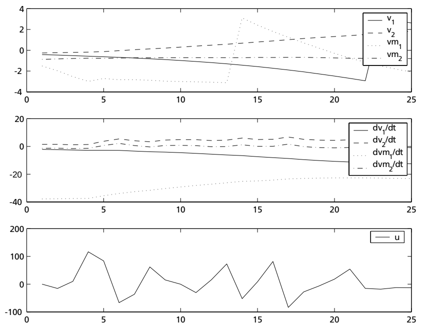
(a) control signal is $u_t = \pm 100$



(b) control signal is $u_t = \pm 100$



(c) control signal is $u_t \in N(0, 100)$



(d) control signal is $u_t \in N(0, 100)$

Figure 2.15: These graphs show simulations of the neural network and the plant. The model is synchronised with the plant at time $t = 0$ and then the plant and model are fed with the same control signal. The resulting states for both the plant and the model are shown. v_1 and v_2 are the angles of the plant, while vm_1 and vm_2 are the angles of the model. Note that the angles wrap around at $\pm\pi$.

This is far beyond the capabilities of the present model. Hence we must conclude that the present approach to construct a neural network model based predictive controller for the acrobot system has failed.

2.4.4 Discussion and Improvements

As noted in the previous section 2.4.3, the prediction done by the model is by far not accurate enough to allow the controller work.

The reason why the model doesn't play nicely is that the plant's behaviour is too complex for the above approach.

A sincere refinement of the model is outside the scope of this thesis, but we will discuss a few thoughts on what could be done to improve the performance.

Improving the Model

In our neural network based model, we're using the neural network to model the plant behaviour. The only pre-processing and post-processing we do is to get zero mean and unit standard deviation on input and output signals.

The neural network will then have to model the underlying function of the plant. We know that this is *possible* since the two-layer neural network we are using can approximate any smooth function for a sufficiently high number of hidden units. But it appears not to be very *efficient.*

If we look at the system model (2.32), we see that it contains a lot of information on how the acrobot state $\mathfrak{y}$ behaves. In our first approach we were not utilising any of this a-priori knowledge.

One solution would be to use an ODE solver to solve equation (2.32) and to use this to predict future plant states.[23]

This way we would leave the neural network based model altogether. There are two things worth noting regarding this solution. First of all, solving an ODE in an on-line system is very time consuming and places a high computational burden on the control system.

[23]ODE: ordinary differential equation.

Another thing is that the ODE solver is fixed, and is not able to model differences between the ODE and the plant.

An in-between solution would be to incorporate some of the information of equation (2.32) into the model.

The state equation (2.32) tells us what the time derivative of the states is. If we introduce Y' as a function of the state Y as follows

$$\begin{aligned} Y'(Y_t) &= T_s \cdot \dot{Y}_t \qquad (2.35) \\ &= T_s \begin{bmatrix} y_3 \\ y_4 \\ \frac{-a_2(y_4^2-2*y_3*y_4)\sin(y_2)+a_4\sin(y_1)+a_5\sin(y_1-y_2)+a_3+a_2\cos(y_2)u}{a_1+2a_2\cos(y_2)} \\ u \end{bmatrix} \end{aligned}$$

this corresponds to a time-wise linear approximation.

The equations for the state one time step ahead can now be expressed as

$$Y_{t+1} = Y_t + Y'(Y_t) + \delta_Y \qquad (2.36)$$

Here δ_Y denotes the mis-match of the linear $Y_{t+1} = Y_t + Y'(Y_t)$ model.

We can now use a neural network to model δ_Y. In this way, we have moved some of the non-trivial a-priori knowledge of kinematic equation (2.32) outside the neural network. But at the same time, our neural network is still able to model mis-match between (2.32) and the plant. A diagram of such a model is shown in figure 2.16.[24]

A different solution would be to calculate only the time derivate of the state y_3 and use this as an auxiliary input variable y_a

$$\begin{aligned} y_a &\equiv \dot{y}_3 \\ &= \frac{-a_2(y_4^2 - 2 * y_3 * y_4)\sin(y_2) + a_4\sin(y_1)}{a_1 + 2a_2\cos(y_2)} \qquad (2.37) \\ &\quad + \frac{a_5\sin(y_1 - y_2) + a_3 + a_2\cos(y_2)u}{a_1 + 2a_2\cos(y_2)} \end{aligned}$$

The variable y_a would then be used as a 6th input to the neural network that is set up in the usual way. This is illustrated in figure 2.17.

[24]Obviously we could move even more information outside the neural network by adding higher order derivatives to the equation (2.36).

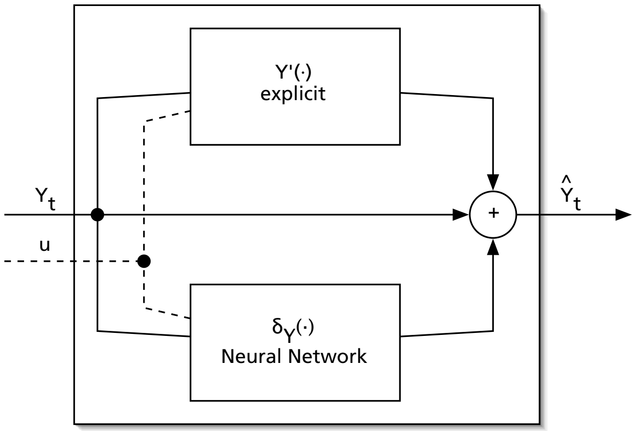


Figure 2.16: Some of the a-priori knowledge is moved outside the neural network, by using a model structure corresponding to equation (2.36) as shown here.

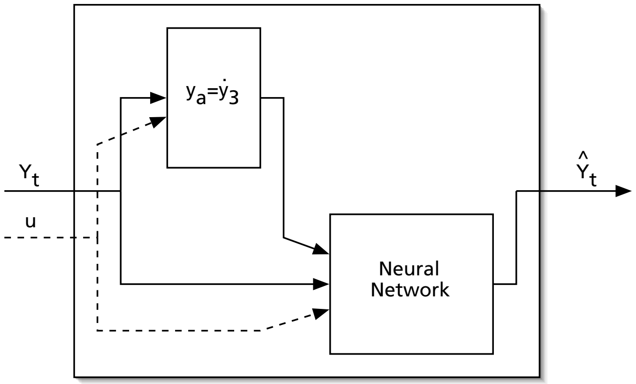


Figure 2.17: A different way of utilising the a-priori knowledge is to feed the time derivatives of the a model structure corresponding to equation (2.36) as shown here.

Since the derivatives of y_1, y_2 and y_4 are given by y_3, y_4 and u respectively, all time derivatives are fed into the neural network in this way – except for a scaling by the sampling time T_s. The scaling is not a problem as this is easily achieved by the neural network. In fact the pre-processing that scales the inputs to unit standard deviation would remove such a scalar anyhow.

2.5 Chapter Discussion

The general predictive control strategy introduced in the beginning of this chapter has an intuitive interpretation and is known to work well for many linear applications.

We have seen how it easily extends to application on non-linear plants. Prediction of future plant outputs is done using a neural network based model of the plant. The projected control series is then optimised based on those predictions.

This extension is mainly troubled by two problems: The process of predicting future plant outputs is non-trivial. While the GPC strategy demands a costing horizon of size comparable to the plant dynamics, the prediction of the plant output may not work over this horizon as was shown in the acrobot implementation in section 2.4.

The second problem is the one of finding an optimal control strategy. While this problem in many cases is merely one of computational speed, other cases such as the coupled tank system in section 2.3 require special care when designing an optimisation algorithm to ensure convergence of the projected control series.

Chapter 3

Direct Neural Network Control

The idea of this controller is very simple. The control diagram is shown in figure 3.1. It is an output feedback controller with a neural network at the core of the controller.

The controller implements some function f on past and present references r and plant outputs y and on past control signals u:

$$u_t = f(r_t, r_{t-1}, \dots, y_t, y_{t-1}, \dots, u_{t-1}, u_{t-2}, \dots) \tag{3.1}$$

Section 3.1 discusses how this controller is constructed using a neural network, and how the controller is tuned to minimise a given cost function.

Sections 3.2 and 3.3 describe two implementations of the direct neural network controller. The first implementation controls a reversing trailer truck to let it follow a given path. The second implements a new controller for the coupled tank system from section 2.3.

3.1 Controller Design

The direct neural network controller we will describe here implements the function (3.1) in the following way:

A regressor function maps the known data at time t into a regressor φ_t of fixed dimension:

$$\varphi_t = \varphi(u_{t-1}, u_{t-2}, \dots, y_t, y_{t-1}, \dots, r_t, \dots)$$

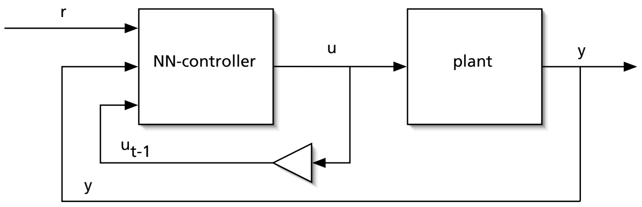


Figure 3.1: The system diagram of a plant controlled directly by a neural network controller. Section 3.1 discusses the design of such a controller.

The regressor φ_t is then used as input to a neural network, and the neural network output is the control signal.

The regressor has to extract the useful information from all known information. A very simple implementation would be to use just a few of the known data values directly, such that e.g.

$$\begin{aligned}\varphi_t &= \varphi(u_{t-1}, u_{t-2}, \ldots, y_t, y_{t-1}, r_t, \ldots) \\ &= [u_{t-1}, y_t, r_t]^T \qquad (3.2)\end{aligned}$$

A more complex approach is to use some linear mapping such as principal component decomposition to remap a larger set of known data to a set of neural network inputs with lower dimension.

In the following we will only use the first method corresponding to equation (3.2).

Next section discusses the number of input data to use as a part of the model order selection followed by section 3.1.2 on neural network training.

3.1.1 Model order selection

The number of neural network outputs is fixed for this controller as the only output is the control signal. Tunable parameters are the number of hidden units and the input units.

The neural network controller we will use has inputs that are the outputs of a lag-network as shown in figure 3.2 since we are using a regressor φ_t of a type as discussed above.

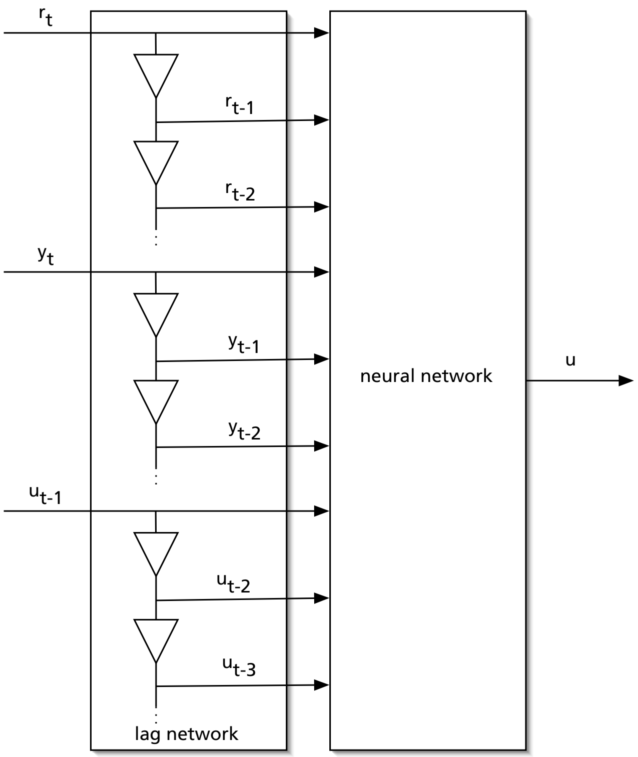


Figure 3.2: The lag network feeds the neural network with current and old signals by storing signals internally

The number of input units hence divides down to three distinct parameters: We can adjust the number of inputs originating from reference signals, plant outputs and control signals respectively.

Determining the number of input signals can be aided by using covariance analysis on time series of the input signals (i.e. r, u and y). This is non-trivial, however, since y and u are feed-back signals that are by nature correlated to themselves.

For the systems that this method was tested on, sparse lag networks with e.g. only one of each signal r, y and u worked just fine. No effort has been made to investigate the results of an increased input signal dimension.

3.1.2 Neural Network Training

Training of the neural network is done by iteratively simulating the system consisting of controller and plant, and then evaluating the resulting time series of plant outputs $(y_1, y_2, \ldots, y_N)$ and control signals $(u_1, u_2, \ldots, u_N)$ with respect to a given cost function J:[1]

$$J = f(y_1, y_2, \ldots, y_N, u_1, u_2, \ldots, u_N) \tag{3.3}$$

An optimisation algorithm is used to minimise the cost function through this iterative process.

It is important to understand how this is fundamentally different from training a neural network using back-propagation. Neural network training as discussed in chapter 2 uses a training set of neural network inputs and outputs (or: targets) and tries to train the network to this data. In that case the error function E during neural network training is evaluated on the neural network outputs, $\tilde{y}_t$, compared to the training data set targets y_t, i.e. $E = \sum f(y_t, \tilde{y}_t)$.

The present neural network training works in a very different way. For a given neural network we run a simulation, and then evaluate how *well* the controller did the job using the cost function (3.3).

This works as follows:

1. run simulation of system
2. evaluate result with respect to given cost function (J)
3. choose new neural network weights according to optimisation algorithm
4. go back to step 1 as long as stop criterion is not met

[1]The cost function will in the general case also include the reference signal, but this has been omitted here for easy readability. The trailer truck discussed in section 3.2 has no reference signal (i.e. $\forall t : r_t = 0$) and the cost function does not have a reference term in that case.

The stop criterion for the optimisation algorithm would be some combination of a maximum number of iterations and some termination tolerance on the cost function.

The optimisation algorithm used in this work is Matlab's `fminsearch`.

Start Guess

The implementations we look at later in this chapter "get away" with a random start guess, but for some systems a decent start guess can be very critical for the optimisation algorithm to converge.

A simple, yet powerful solution to finding a decent start-guess is to use a linear controller and approximate this controller with a neural network. In this way we obtain a start guess for the neural network weights.

First we design a linear controller (e.g. LQG) using a well-known design method on a linear approximation of the plant. This controller can be implemented as

$$\begin{aligned} u_t &= \begin{bmatrix} w_1 & w_2 & w_3 & \cdots \end{bmatrix} \\ &\quad \begin{bmatrix} r_t & r_{t-1} & \cdots & y_t & y_{t-1} & \cdots & u_{t-1} & u_{t-2} \end{bmatrix}^T \end{aligned} \tag{3.4}$$

Then we use a very simple network structure with one hidden unit, no biases and input units corresponding to the signals available in the linear controller.

A network with only one hidden node, one output and no bias has the following transfer function (Cf. equation 1.5):

$$x = W \cdot \tanh\left(\sum_{i=0}^{d} w_{ji} p_i\right)$$

For small values, the hyperbolic tangent can be approximated by a linear function (Cf. figure 1.2). This allows us to approximate the neural network transfer function as follows for small activations (sum of inputs) of the hidden node:

$$x \approx W \sum_i (w_i p_i) \tag{3.5}$$

We can now take the values $w_1, w_2, \ldots$ from (3.4) and use as neural network weights and set

$$W = 1$$

To ensure that the activation (sum of inputs) of the hidden node is not too far away from the approximately linear region, we can scale the neural network weights W and w with some scalar k:

$$w' = \frac{w}{k} \qquad W' = kW$$

The resulting neural network will be a good start guess.

Changing the neural network model to another order is straight forward. Adding input nodes with zero weight or *dividing* the hidden node(s) with corresponding scaling of the neural network weights W and w is trivial.

3.2 Implementation of the Reversing Trailer Truck

The parametric optimisation algorithm will now be used on a reversing trailer truck. The controller is steering a reversing truck that pushes a trailer along a given path. We will first define a coordinate system that eases the subsequent kinematic system description.

3.2.1 System Description

The problem of reversing the trailer truck along a path is best described in the coordinate system of the path. Terms such as *"along the path"* and *"close to the path"* are complicated to express in a Cartesian (x, y)-coordinate system.

Coordinate System

Referring to figure 3.3 we therefore introduce a new (s, d)-coordinate system as done in [8].

The coordinates of the point P_0 are defined as follows: Let the path to be followed be denoted by C. For the point P_0, let $P_{0,\text{proj}}$ be the point on C

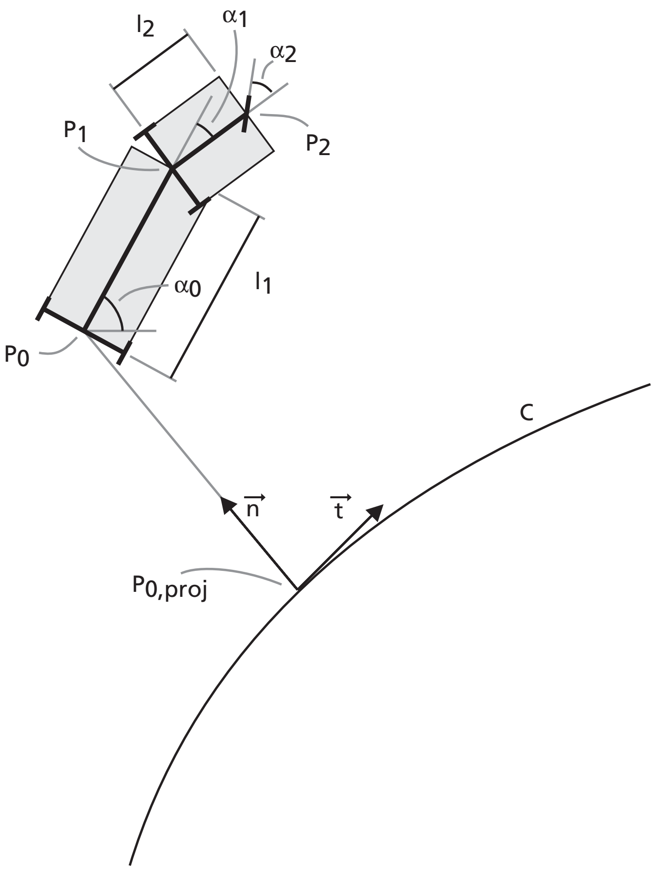


Figure 3.3: The path-relative coordinate system and the trailer truck. $\vec{t}$ is the tangent of the curve C at the point $P_{0,proj}$, and $\vec{n}$ is the normal at that point. Angles and physical parameters of the truck are also shown.

closest to P_0. Now the s coordinate is defined by the distance along C from the beginning of C (where $s = 0$). d is defined as the scalar that fulfils

$$\overrightarrow{P_{0,\mathrm{proj}} P_0} = d\vec{n}$$

i.e. d is the signed distance from $P_{0,\mathrm{proj}}$ to P_0.

Note that the mapping $(s, d) \to (x, y)$ is unique while the mapping $(x, y) \to (s, d)$ may be ambiguous.

We will assume that there is a lower bound $r_{\min} > 0$ to any circle tangenting C at two or more points and the interior of which does not contain any point of the curve.

This assumption limits how close two segments of the path can be to each other. This implies that the path is not allowed to cross itself. The parameter $r_{\min}$ also puts a limit on how tight turns the path may take. The bound $r_{\min} > 0$ implies, for the curvature $\mathrm{curv}(s)$ that

$$\forall s \in [0; S] : \mathrm{curv}(s) \leq \frac{1}{r_{\min}} \tag{3.6}$$

where S denotes the path length.

To avoid any ambiguity in the parameterisation one of the control objectives is to keep the coordinate d smaller than $r_{\min}$ at all times.

In this coordinate system following the path means that we want to keep d close to 0. We will interpret the angle θ in such a way that θ is close to 0 when we are reversing along the path.

Kinematic equations

The physical parameters of the trailer truck are depicted in figure 3.3 and listed in table 3.1.

The kinematic equations of the trailer truck are derived in [8]. We shall shortly summarise the results below. Note that $\mathrm{curv}(s)$ is the curvature of the path at point s.

The kinematic equations describing the trailer truck motion are:

$$\dot{s} = v_0 \frac{\cos(\theta)}{1 - \mathrm{curv}(s)d}$$

l_i	distance between P_i and P_{i+1}
α_0	angle of the first vehicle with respect to a fixed frame. We choose the Cartesian base coordinate system as reference for this angle.
$\alpha_i \quad (1 \leq i \leq n)$	angle between $P_{i-1}P_i$ and P_iP_{i+1} (the orientation of vehicle $(i+1)$ with respect to the previous vehicle)
α_{n+1}	angle of the car's driving front wheel with respect to the car's body.
v_i	intensity of the velocity of the point P_i. This is the translational velocity of the $(i+1)$ vehicle.

Table 3.1: The physical parameters of the trailer truck. See figure 3.3 for illustration.

$$\begin{aligned}
\dot{d} &= v_0 \sin(\theta) \\
\dot{\theta} &= v_0 \frac{\tan(\alpha_1)}{\ell_1} - \frac{curv(s)\cos(\theta)}{1 - curv(s)d} \qquad (3.7) \\
\dot{\alpha_1} &= v_0 \frac{1}{\cos(\alpha_1)} \left(\frac{\tan(\alpha_2)}{\ell_2} - \frac{\sin(\alpha_1)}{\ell_1} \right) \qquad (3.8) \\
\dot{\alpha_2} &= u
\end{aligned}$$

and since

$$v_i = v_{i+1} \cos(\alpha_{i+1}) \qquad (0 \leq i \leq n)$$

this can be rewritten as

$$\begin{aligned}
\dot{s} &= v_2 \cos(\alpha_1)\cos(\alpha_2) \frac{\cos(\theta)}{1 - curv(s)d} \\
\dot{d} &= v_2 \cos(\alpha_1)\cos(\alpha_2)\sin(\theta) \\
\dot{\theta} &= v_2 \cos(\alpha_2) \frac{\sin(\alpha_1)}{\ell_1} - \frac{curv(s)\cos(\theta)\cos(\alpha_1)}{1 - curv(s)d} \qquad (3.9) \\
\dot{\alpha_1} &= v_2 \left(\frac{\sin(\alpha_2)}{\ell_2} - \frac{\sin(\alpha_1)\cos(\alpha_2)}{\ell_1} \right) \\
\dot{\alpha_2} &= u
\end{aligned}$$

The control signal $u = \dot{\alpha}_1$, i.e. the control signal controls the change in the truck's driving front wheel's angle.

We will assume that the speed is constant. As long as there is no limit on our control signal $u = \dot{\alpha}_1$ this makes sense.

A change in speed is needed when $\dot{\alpha}_1$ can not exceed a given limit. We would then need to decrease the truck speed in order to change the angle α_1 fast enough relative to the translational movement.

We will stick to the simple case, where $u = \dot{\alpha}_1$ is not limited.[2]

3.2.2 Bezier Path Implementation

To simulate the reversing trailer truck, we need a mathematical description of the path. Throughout this section we use x and y to refer to Cartesian coordinates – not states or other signals.

The path that the trailer truck is to follow is implemented as a Bezier curve. We've chosen Bezier curves since they have a well defined curvature and use an intuitive parameterisation. It is important to have a well defined curvature, both due to the constraint (3.6) and due to the fact that the kinematic equations (3.9) are based on the path curvature. The intuitive parameterisation furthermore makes it simple to construct paths to our liking.

Splines

The Bezier path consist of several splines that are concatenated. The spline is a parametric curve p parameterised by the free variable $t \in [0; 1]$:

$$p(t) = (x(t), y(t)) \tag{3.10}$$

For the Bezier spline, the functions x and y are given by[3]

$$p(t) = \sum_{i=0...3} B_i(t) p_i$$

[2] It is not entirely true that there is no limit on u. We are penalising control signal increments and thus also a change in control signal needed to obtain high control signals. We will ignore this, though, since the control cost has been set to a low level.

[3] $B_i(t)$ denotes the i'th Bernstein polynomial of t.

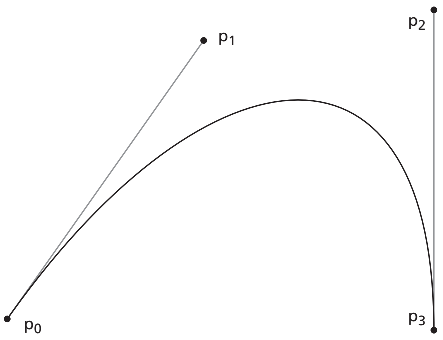


Figure 3.4: The points p_0, p_1, p_2 and p_3 define the Bezier spline. The point p_0 is the starting point of the spline, and p_3 is the end point of the spline. The points p_1 and p_2 define how the spline forms from p_0 to p_3. The tangent in p_0 is equal to $p_0\vec{p}_1$ and correspondingly for p_3 and p_2.

$$B_i(t) = \binom{3}{i} t^i (1-t)^{3-i}$$

$$p(t) = (1-t)^3 p_0 + 3(1-t)^2 t p_1 + 3(1-t)t^2 p_2 + t^3 p_3 \quad (3.11)$$

such that

$$x(t) = (1-t)^3 x_0 + 3(1-t)^2 t x_1 + 3(1-t)t^2 x_2 + t^3 x_3 \quad (3.12)$$
$$y(t) = (1-t)^3 y_0 + 3(1-t)^2 t y_1 + 3(1-t)t^2 y_2 + t^3 y_3 \quad (3.13)$$

The points p_0, p_1, p_2 and p_3 are shown in figure 3.4 with the corresponding Bezier spline.

The above parameterisation uses $t \in [0; 1]$ as the free parameter. The coordinate system defined in section 3.2.1 is based on the distance s along the path. We have to change the parameterisation of (3.11) from t to s.

Mapping the variable t to s is done by using the arc length parameterisation[4]

$$s(t) = \int_0^t \sqrt{p'(\tau) \cdot p'(\tau)} d\tau$$

[4]Here p', x' and y' are the derivatives with respect to τ.

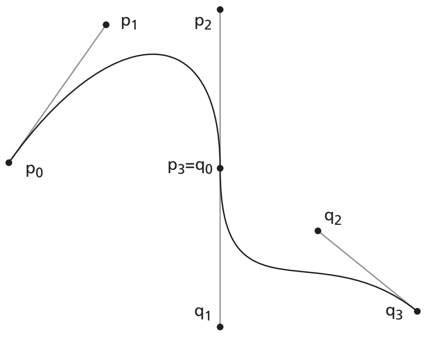


Figure 3.5: The Bezier curve is made up of several splines fitted together end to end as shown here. The point p_2 of the old spline is mirrored through the new starting point q_0 to obtain the point q_1.

$$= \int_0^t \sqrt{x'^2(\tau) + y'^2(\tau)} d\tau \qquad (3.14)$$

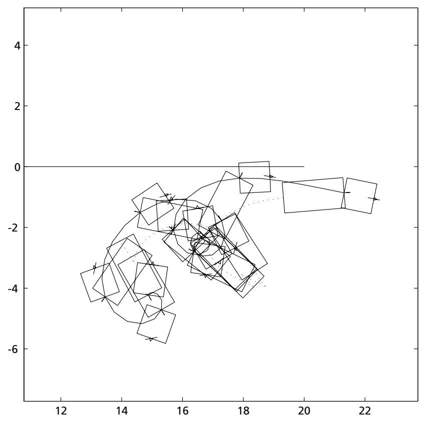
Note that $s(t)$ is monotonic, and hence the inverse function $t(s)$ is uniquely defined.[5]

We implement the inverse function by using a table of values for the $s(t)$ function and doing reverse lookup in this table. With sufficient data sets in this table and using linear interpolation an arbitrary level of approximation can be achieved.

As noted the Bezier curve will consist of several Bezier splines fitted together end to end. The end point p_3 of one spline will be the starting point q_0 of the next. The point p_2 of the old spline will be mirrored through the new starting point to obtain the new q_1. In this way, the tangent at the end point of the old spline will be the same as the tangent at the starting point of the new spline. The resulting Bezier path will have a well-defined curvature. This is illustrated in figure 3.5.

Figure 3.12 on page 73 shows the Bezier path used for training the network. The division of the path into splines is apparent. On other paths

[5]We are using this loose syntax to ease readability. $s(\tau) = s|_{t=\tau}$ and $t(\sigma) = t|_{s=\sigma}$.


(a) Meaningless movement going nowhere

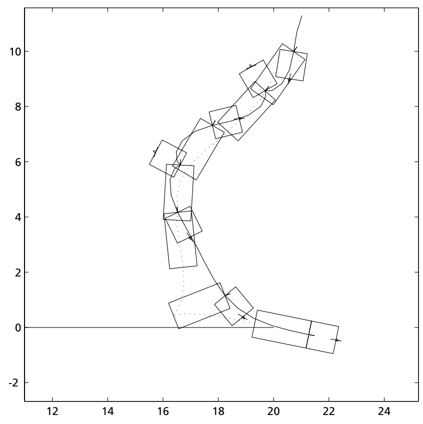

(b) The truck is spinning and pulls the trailer instead of pushing it.

Figure 3.6: The trailer truck movement with a network structure as shown in figure 3.8(b) and random weights. Both examples show how the truck and trailer overlap.

such as the one shown in figure 3.18 on page 78 the start and end points of splines are less obvious.

3.2.3 Training the Neural Network

The control problem of the reversing trailer truck consists of two interrelated problems. One problem is to follow the path. The other one is the unstable problem of reversing the truck without the trailer and the truck getting entangled as the angle α_1 grows above 90°.

Starting off with the complete problem it is impossible to get the optimiser to converge. The problem is too complex. To give an idea of this problem figure 3.6 shows the trailer truck movement for a random network with a network structure as shown in figure 3.8(b). Therefore we approach the problem with a two-stage solution.

Reversing the Trailer Truck

First we try to train the network to be able to reverse the trailer truck in a given direction without the truck and trailer colliding (i.e. we keep $\forall t : \alpha_1 < \frac{\pi}{2}$). Then we'll train the network to reverse the trailer truck along the path.

During the first step, we disregard the distance of the trailer to the path, and we choose a straight line as the path to follow. We want the trailer truck to go in this line's direction while keeping the trailer and truck from colliding.

To achieve this , we use the following cost function

$$J_1 = s_{end} + \frac{1}{N}\sum_{i}^{N} \theta_i^2 + \alpha_1^2 + \left(\frac{\alpha_1}{0.95 \cdot \pi/2}\right)^{40} + \left(\frac{\alpha_2}{0.95 \cdot \pi/2}\right)^{40} \quad (3.15)$$

where s_{end} is the total distance travelled along the path. This term favours network weights that get the trailer moved far along the path.

In the summation the first two terms are regular quadratic cost terms on the angels θ and α_1. These are supposed to tell the optimiser to let the trailer truck straighten up and go in the same direction as the path.

The last two terms inside the summation are penalty terms. These terms keep the angles α_1 and α_2 from going outside the interval $]-\frac{\pi}{2}; \frac{\pi}{2}[$. Figure 3.7 shows a plot of the penalty function.

At the same time we use a simple network structure as shown in figure 3.8(a) where only the three angles θ, α_1 and α_2 are fed into the neural network controller.

As noted in section 3.1.2 training of the neural network is done by running a simulation of the complete system and evaluating the cost function (in this case equation (3.15)) on this simulation.

As figure 3.6 shows, these simulations will lead to meaningless movement for some weight combinations. Therefore termination conditions for the simulations need to be chosen with care.

The simulations will be terminated if either the end of the path is reached: $s_{end} \geq S$ or a given maximum iteration count has been reached.

The neural network is then trained with a random start guess. We use 4 training scenarios. In the first scenario the trailer and truck point in the

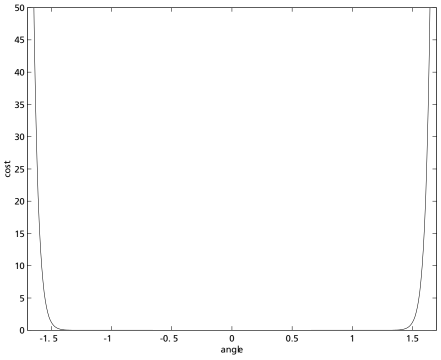


Figure 3.7: The penalty function on the angles α_1 and α_2 in the cost functions (3.15) and (3.16) is used to penalise angles outside $]-\frac{\pi}{2};\frac{\pi}{2}[$. Note that $\frac{\pi}{2} \approx 1.57$.

same direction ($\alpha_1 = 0$), in the second the trailer is angled with respect to the truck ($\alpha_1 = 0.2$). In both scenarios the trailer truck is angled with respect to the direction it has to follow.

Figures 3.10 and 3.11 on pages 70 and 71 show these two start scenarios at the very right hand side of the plots.

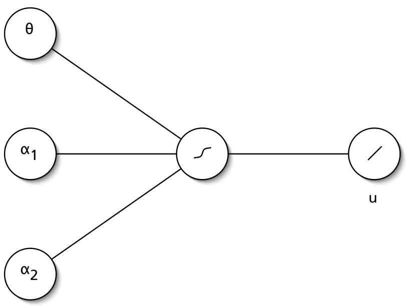
The last two scenarios are the mirrored versions of the two first (all angles are mirrored). The presence of these mirrored scenarios ensures that the network is not biased to favour a turn in a certain direction (e.g. that it will have a tendency to always turn to the right).

The cost function 3.15 is evaluated on the four simulation runs of the four scenarios.

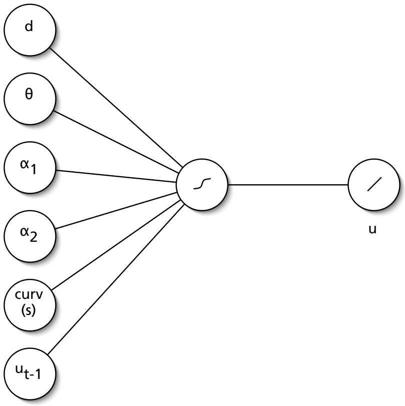
After approximately 200 iterations of the optimiser, the neural network is able to reverse the trailer truck in the right direction and straighten it up.

The resulting network with weights is illustrated in figure 3.9. Note that the output weight is fixed to 10.

The movement of the reversing trailer truck and the signals are plotted in figure 3.10 and figure 3.11 for two of the four scenarios. Note that the trailer truck motion is right to left, while the signals are plotted left to



(a) The reduced neural network structure used for initial training of the network.



(b) The full neural network structure, that controls the reversing trailer truck.

Figure 3.8: The controller is trained using a two step approach. First the the simpler network structure in figure 3.8(a) is used with cost function (3.15) to train the network to be able to reverse the truck properly. Then the full network shown in figure 3.8(b) is used with the cost function (3.16) to control the complete system.

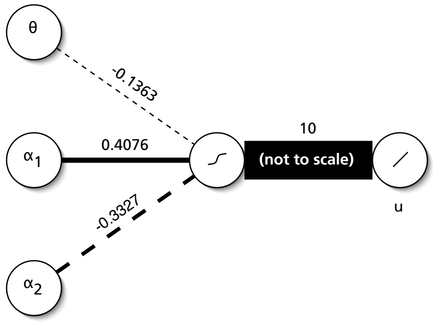


Figure 3.9: The resulting weights after training of the neural network controller to straighten up the reversing trailer truck. Line widths are relative to absolute weights (except for the weight of 10). Dashed lines indicate negative values.

right.[6]

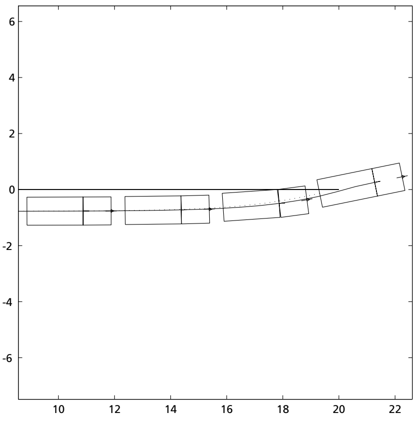
Following the Path

Once the network is able to reverse the trailer truck we want the neural network controller to learn to keep the trailer truck on the path.

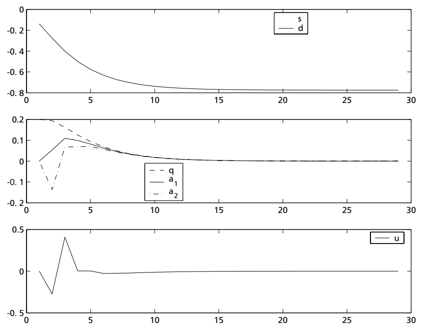
The fact that we want the reversing trailer truck to follow the path has to be expressed by means of a new cost function. We use the following new cost function

$$\begin{aligned} J_{\text{truck}} &= \frac{1}{N}\sum_{i}^{N}\Bigg[\rho_d d^2 + \rho_\theta \theta_i^2 + \rho_\alpha \alpha_1^2 \\ &\quad + \left(\frac{\alpha_1}{0.95\cdot\pi/2}\right)^{40} + \left(\frac{\alpha_2}{0.95\cdot\pi/2}\right)^{40}\Bigg] \qquad (3.16) \\ &\quad + \rho_u \frac{1}{N}\sum_{i}^{N}(\Delta u)^2 \end{aligned}$$

[6]The other two scenarios are mirrored versions of these two with all angles and the start offset reversed as discussed earlier.

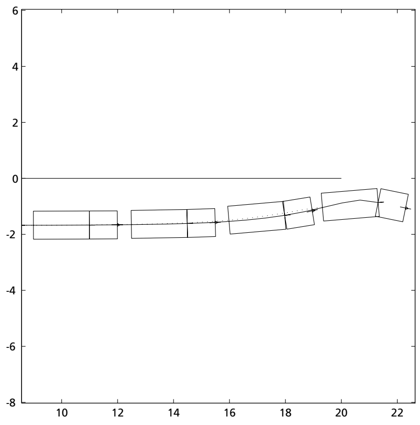


(a) Motion of the trailer truck

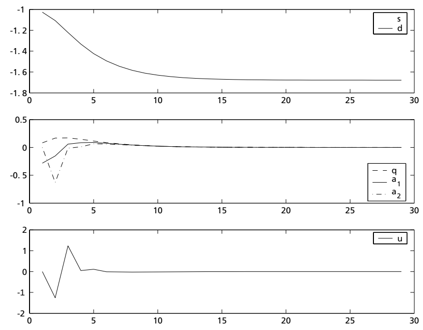


(b) Signals

Figure 3.10: After initial trailing with the cost function 3.15 the reversing trailer truck straightens up from an angled start position with the truck and the trailer pointing in the same direction. The truck and trailer outlines are plotted with distances corresponding to 5 sampling periods. The path of the truck and the trailer rear are plotted as solid and dotted lines respectively. Note that the truck motion in 3.10(a) is right to left while data is plotted left to right in 3.10(b).



(a) Motion of the trailer truck



(b) Signals

Figure 3.11: As in figure 3.10 the controller successfully controls the reversing trailer truck to follow the direction of the x-axis. In the start position, the trailer and the truck are angled. Note again, that the truck motion is right to left while data is plotted left to right.

We've chosen the following cost weights

$$\rho_d = 10 \qquad \rho_\theta = 0.2 \qquad \rho_\alpha = 0.2 \qquad \rho_u = 0.001$$

When compared with (3.15), the new terms are quadratic costs on the distance to the path d and on the control signal change Δu. Furthermore we have added weights on the quadratic costs to balance control aims.

We will now extend the neural network structure to the one shown in figure 3.8(b) on page 68. All parameters

$$d, \theta, \alpha_1, \alpha_2, u_{t-1}$$

will be fed into the network.[7]

The training path is shown in figure 3.12 and was chosen as a combination of 90° turns with different curvatures. The curvatures of the training path have to span the interval of curvatures we want the trailer truck to handle on new paths.

While neural networks are known to be good at interpolating, extrapolating of data in general does not work well with neural networks. Hence the curvatures of the training path have to span the interval of curvatures we want the trailer truck to handle on new paths.

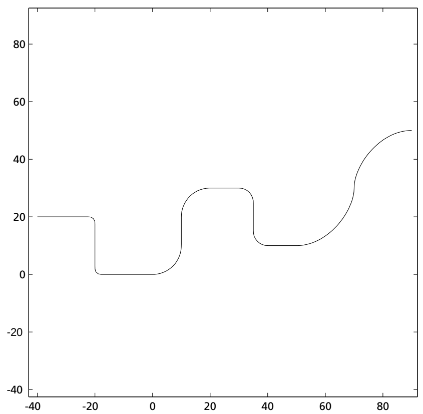
To ensure that the training path is balanced, each right turn is followed by a left turn of the same curvature. This keeps the network from favouring right turns over left turns and vice versa.

The neural network is trained with these new settings, and the resulting network weights are shown in figure 3.13. The performance of the neural network controller is discussed in the next section 3.2.4.

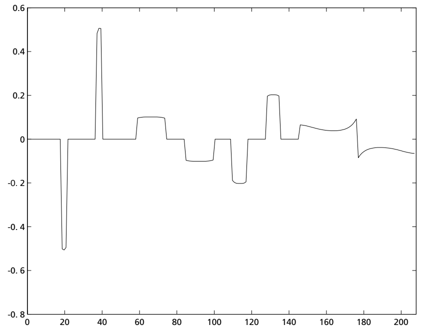
3.2.4 Performance

The neural network controller is trained as described above in section 3.2.3. The resulting motion along the training path (figure 3.12 on page 73) is shown in figure 3.14. Two enlarged areas are shown. The signals for the motion along the entire path are shown in figure 3.15.

[7]There is no point in feeding the 5th model parameter s into the network. s is the distance along the path and holds no information on control actions to be taken.



(a) The training path



(b) The curvature of the training path

Figure 3.12: The Bezier path used for training and the path curvature along the path. Alternating right and left turns ensure, that training is balanced. The curvature of the turns is chosen to span from zero up to 0.5.

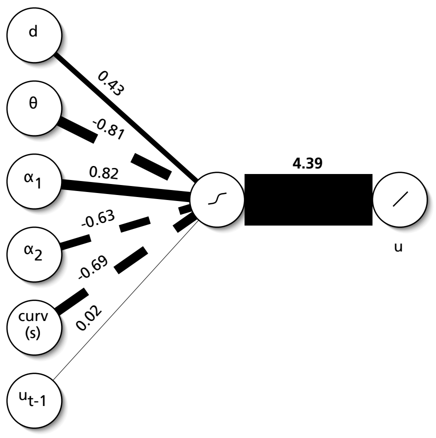


Figure 3.13: The resulting weights after training of the neural network controller for path following of the reversing trailer truck. Line widths are relative to absolute weights. Dashed lines indicate negative values. Cf. figure 3.9.

The trailer truck outlines shown are plotted every 5 sampling times. The *edges* on the motion trails are due to the sampling. Motion data is only sampled for each controller step. It is apparent that the controller has problems with stabilising the movement.

The trailer truck keeps moving from one side of the path to the other and back again. This becomes even clearer when looking at figure 3.15. The distance d to the path is oscillating around 0.

The reason for this is discussed below, and a new neural network controller is trained to fix this problem. It turns out that the last two turns of the training path are very sharp (i.e. have a sudden, high change curvature), and that the controller weights have to be very high in order not to *loose* the trailer truck off the path in those turns.

These weights, however, are not optimal for paths that do not have this kind of turns, but are smoother.

Overall the controller is able to keep the reversing trailer truck on the path.

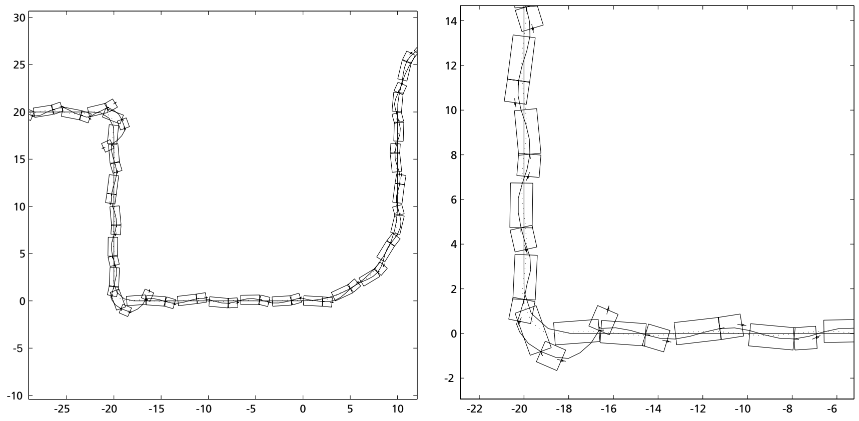


Figure 3.14: Enlarged areas of the reversing trailer trucks motion along the training path shown in figure 3.12 on page 73.

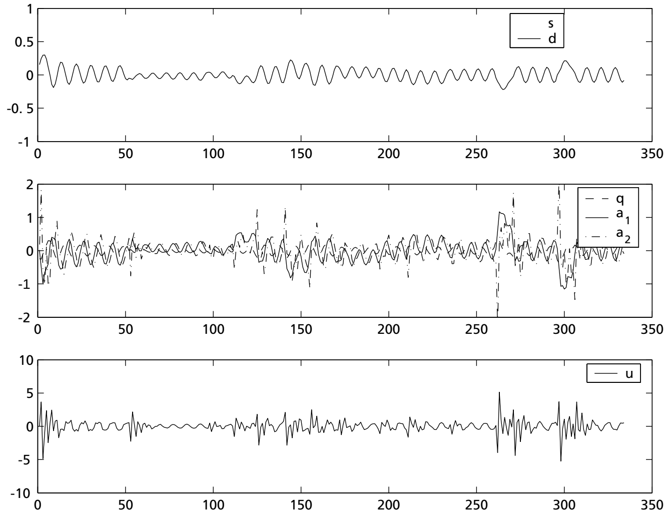
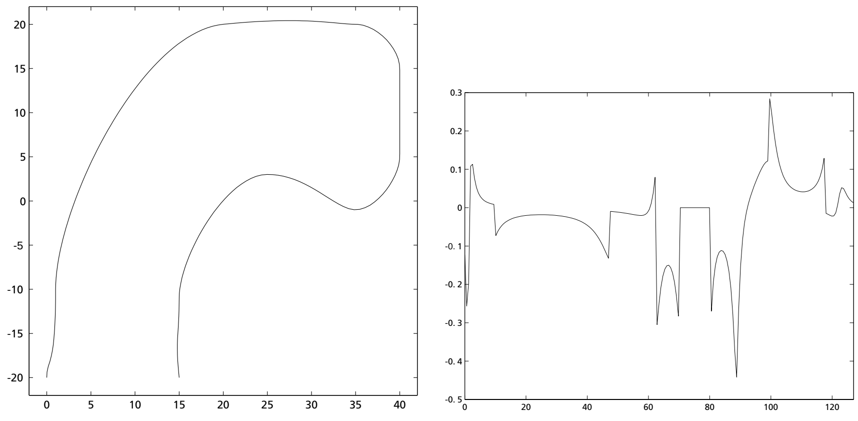


Figure 3.15: The signals while following the training path. Cf. figure 3.12 and 3.14



(a) The path (b) The path's curvature

Figure 3.16: A Bezier path with curvatures equal of magnitude as the training path shown in figure 3.12

Ability to Follow New Paths

We will now see how the controller performs on a new path. Figure 3.16 shows a path that has curvatures of equal magnitude as compared with the training path.

We now run controller and reversing trailer truck on this path. Simulation results are shown in figure 3.17. The trailer truck also follows this path and the problems noted above remain. The controller keeps the trailer truck close to the path, but the distance is oscillating.

The reason for this is discussed below when the network is re-trained.

Parameter Sensitivity

The fact that the controller doesn't perform optimal on paths with lower curvature becomes apparent when we look at the change in cost function as a result of a change in neural network weights when we evaluate the cost function on simulations runs on the path shown in figure 3.18 with

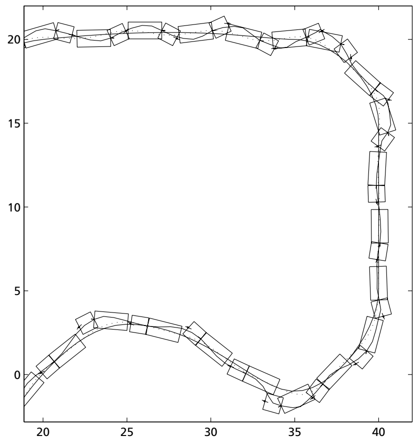


Figure 3.17: The motion of the reversing trailer truck on the new path shown in figure 3.16

moderate curvatures.

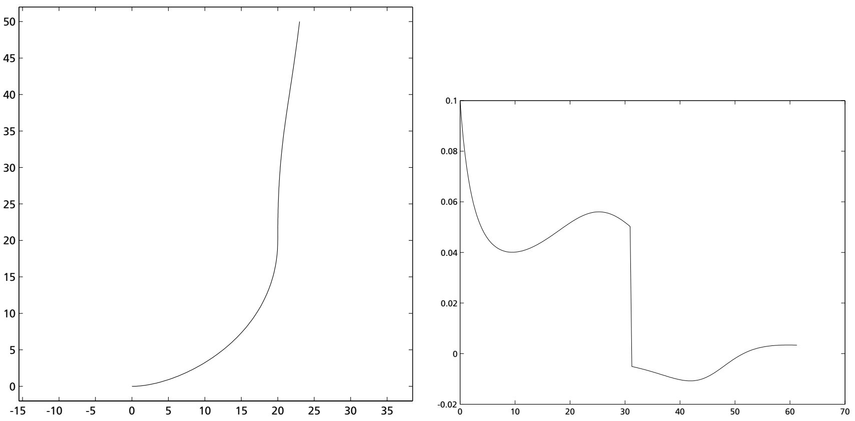
We have evaluated the sensitivity of the system to change in weights of the neural network on the one hand and in change of the physical parameters of the trailer truck on the other.

We have used the cost function (3.16) on simulation runs on the path shown in figure 3.18. The results are shown in figure 3.19.

The sensitivity of the controller towards changes in physical parameters shown in figure 3.19(a) is not very spectacular. If the ratio of the truck length l_1 to the trailer length l_2 increases, the controller performance decreases. The change in v_2 corresponds to the truck moving faster than the speed, the controller *thinks* the truck is moving.

Of more interest is the second figure 3.19(b), showing the sensitivity towards change in network weights. What we would expect is that neural network weights are optimal and hence all change – no matter in what direction – would decrease the controller performance. What we see, however, is that the controller performs better if weights are changed away from the trained values.

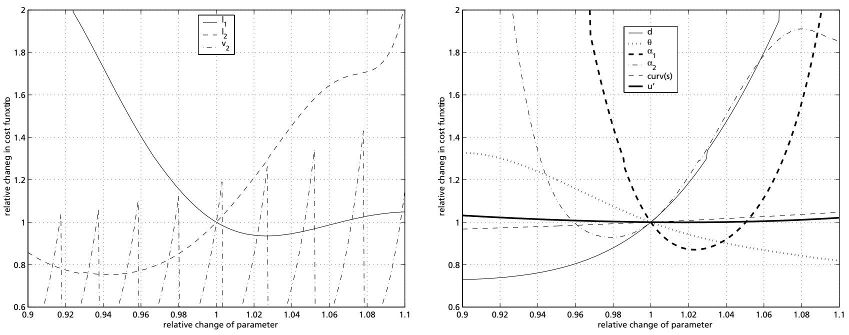
As noted above, this is due to the fact that the training path has some very



(a) The path

(b) The curvature of the path

Figure 3.18: This Bezier path was used for evaluating the parametric sensitivity of the controller trained on the path shown in figure 3.12



(a) Sensitivity on the physical parameters

(b) Sensitivity on the neural network weights

Figure 3.19: The change in cost function as the result of a change in either physical parameters or weights of the neural network controller is shown here. The cost function is evaluated on the path shown in figure 3.18. Note how a change can actually reduce the cost function. This is due to the fact that the network is trained on a different path – the one shown in figure 3.12. This is discussed further in the text.

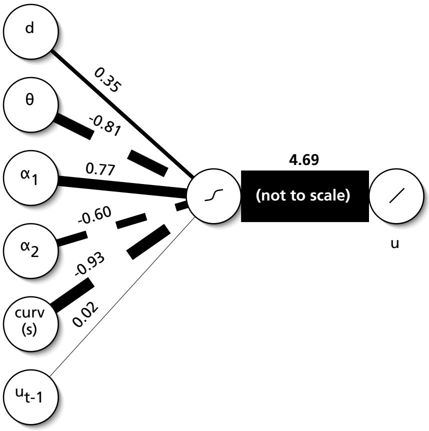


Figure 3.20: After re-training the neural network controller with the new training path weights are as illustrated. Line widths are relative to absolute weights. Dashed lines indicate negative values. Cf. figure 3.13.

sharp turns. The network weights resulting from training are chosen in a way that keeps the system from entering a state where the penalty functions of the cost function (3.16) are activated.

3.2.5 Re-training the Neural Network

Performance on paths with moderate curvature was sub-optimal, and the reversing trailer truck was zig-zagging along the path due to the harsh requirements of the original training path in figure 3.12.

To improve the system performance on paths with moderate curvature, we use a new training path by removing the two sharp turns from the original training path, and then retrain the neural network controller.

Training is done as described in the second part of section 3.2.3 with the mentioned cost function (3.16). The resulting network weights are shown in figure 3.20.

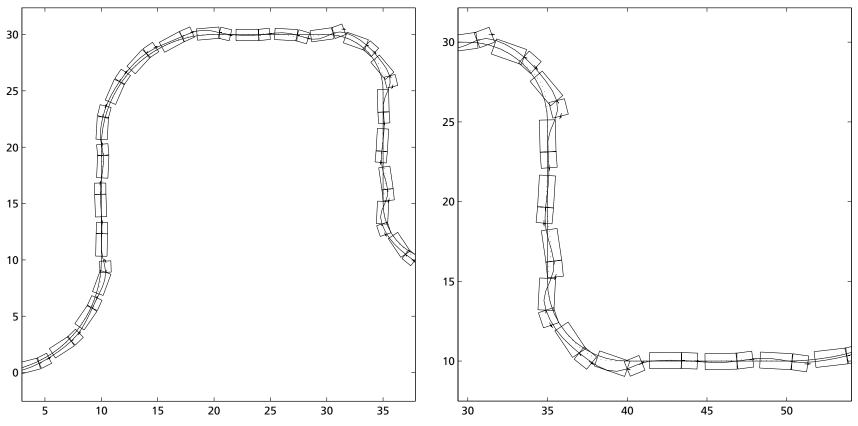


Figure 3.21: Enlarged areas of the reversing trailer trucks motion along the new training path.

3.2.6 Performance, revisited

The motion of the reversing trailer truck along the new training path is shown in figure 3.21 and figure 3.22.

The improvement over the results illustrated in figure 3.14 and 3.15 are apparent. The oscillation of the motion is damped, and the reversing trailer truck is in average closer to the path.

To further demonstrate this, the parameter sensitivity analysis is re-run.

It is apparent from figure 3.23(b) that the new neural network weights are closer to an optimum for this path.[8] Also the sensitivity to change in physical parameters has been reduced.

3.2.7 Discussion

The neural network based direct controller is able to control the reversing trailer truck.

[8]Note that the parameter sensitivity is evaluated on the path shown in figure 3.18 and not the training path.

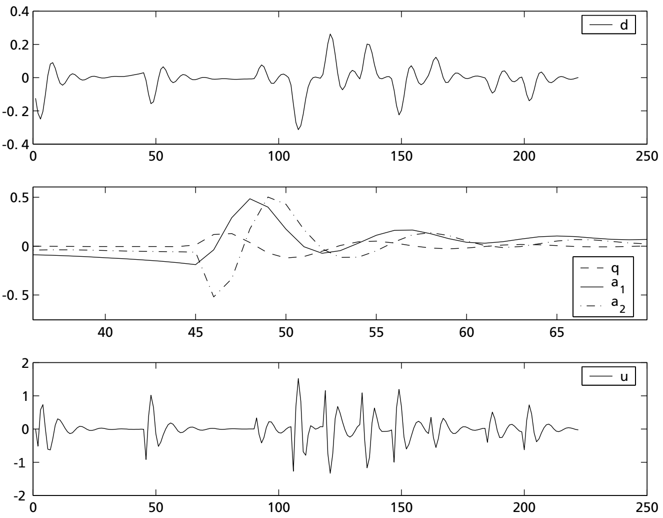
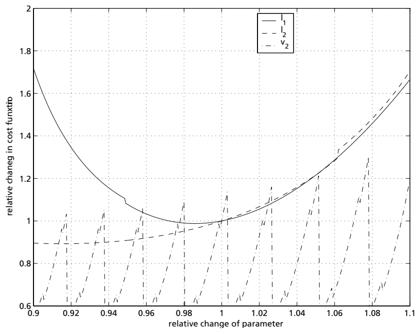
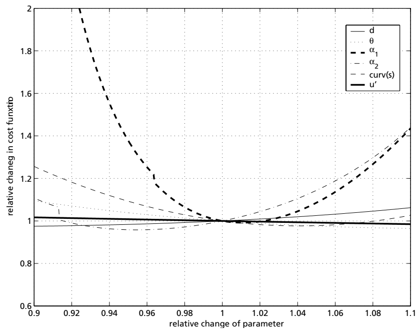


Figure 3.22: The signals while following the new training path. Cf. figure 3.21.



(a) Sensitivity on the physical parameters



(b) Sensitivity on the neural network weights

Figure 3.23: The parameter sensitivity as shown in figure 3.19 on page 79, but this time after additional training of the neural network controller on the new training path.

Using a path based coordinate system, transformation of the control goal is simple. Using a random start guess, the tuning of the controller is done in a two-staged approach. First the controller *learns* to reverse the trailer truck, and only then it is faced with the full control problem.

The optimisation of the controller parameters is done in a *brute force* way by using a simplex optimiser on the parameters and a cost function on the simulation runs.

Since the controller is a single step ahead controller, the plant output has a tendency to have damped oscillations.

One of the reasons that the controller works well is that the plant has a local linear approximation.

3.3 Implementation of the Coupled Tank System

We will now use the direct neural network control on the coupled tank system described in section 2.3.1.

In the last chapter this plant was controlled with a neural network model based predictive controller. We will now see, how the present controller compares with the one from chapter 2.

Opposed to the reversing trailer truck system, the coupled tank system has a reference signal. This fact changes things only slightly. The process of training the neural network and the resulting performance are discussed in the following.

3.3.1 Neural Network Training

The neural network we will use has 4 inputs: The states $Y_{1,t}$, $Y_{2,t}$, the reference signal r_t, and the last control signal u_{t-1}.

We are using two hidden units. There's only one network output, the control signal u_t. The network structure is shown in figure 3.25 with the weights after training.

We will train the neural network to minimise the cost function

$$J = \frac{1}{N} \sum_{t}^{N} (y_t - r_t)^2 + \rho(\Delta u_t)^2 \qquad (3.17)$$

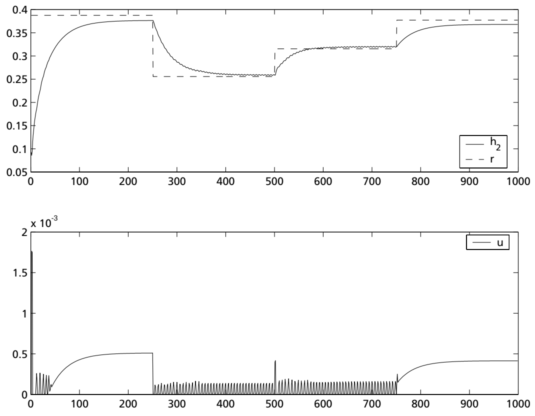


Figure 3.24: The reference signal used for training the neural network is a N-samples-constant signal that changes every 250 samples and has a total length of 1000 samples like the one shown here.

During training, an N-samples-constant reference signal with a total length of 1000 samples is used. This signal changes every 250 samples and is constant in between these changes. This allows the controller to let the plant output reach the reference before the next change in reference. Figure 3.24 shows one such reference signal.

The neural network weights are now optimised for 40 iteration steps using a reference signal like this one, and then a new reference signal is chosen. This helps generalisation without increasing the computational burden of each iteration of the optimiser.[9]

As noted above, negative control signals can not be implemented by the coupled tank system. The control signal is therefore limited to $\forall t : u_t \geq 0$. For this reason if the control signal generated by the controller is negative, it is reset to $u_t = 0$.

The network weights obtained by training are shown in figure 3.25.

[9]Another way to get good generalisation would be to use simulation runs with more samples. But this would decrease the speed of the network weight optimisation.

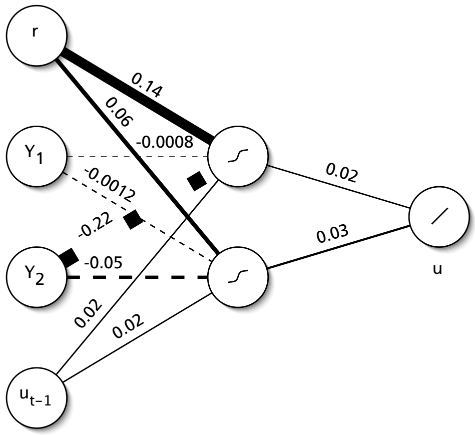


Figure 3.25: The neural network with the weights. Line widths correspond to weights, and negative values are indicated by dashed lines. Note how the reference signal r and the tank level difference Y_2 have the greatest impact on the hidden node activation.

3.3.2 Performance

To test the performance of the controller, simulation runs equal to the ones in chapter 2 were done. The results are shown in figure 3.26.

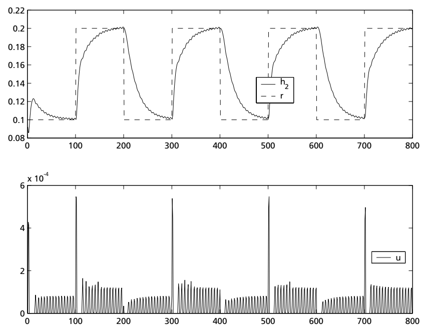
What immediately catches the eye is that also this controller has problems with the constraint $u > 0$. The control signal oscillates when it is small.

The controller tries to set a negative control signal, but the control signal is limited to $u = 0$. The result of this *confuses* the controller as it sets a higher value the next time and starts to oscillate.

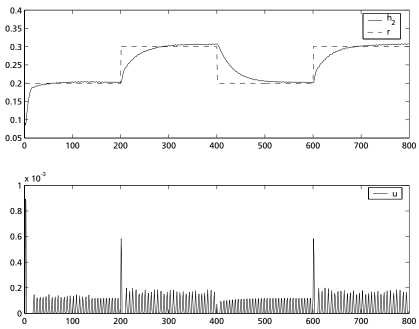
Less visible is the fact, that the controller has a stationary offset. If we turn back to figure 3.24 on page 84, we will realise that this behaviour is due to the way the neural network was trained.

Figure 3.24 shows that the main source of error with respect to the cost function (3.17) is deviation of the plant output y from the reference r immediately after the change of the reference.

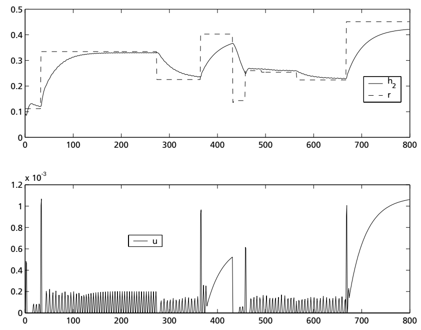
The term $(y_t - r_t)^2$ is very small once the plant output is close to the



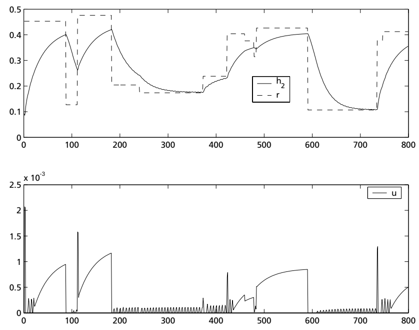
(a) reference is square wave with period of 100T$_s$



(b) reference is square wave with period of 200T$_s$



(c) reference is a level-change-at-random-instances signal



(d) reference is a level-change-at-random-instances signal

Figure 3.26: Simulation runs of the direct neural network controller. Cf. figure 2.9 on page 36.

reference, and hence the optimiser will favour a controller that will make the plant output reach the vicinity of the reference within a short time over a controller that has a small stationary offset.

3.3.3 Discussion

As opposed to the predictive controller for the coupled tank system implemented in chapter 2, the present controller is a single-step ahead controller. This shows by the oscillating behaviour on the one hand, and on the other hand that the controller does not act upon a level change of the reference, before the reference change is reached.

This said, the control signal of this controller is deterministic and does not contain noise. The predictive controller in chapter 2 generated a control signal that contained noise that was introduced by the poor convergence of the optimisation algorithm.

It is likely that extending the neural network structure to a higher order, would prevent the mentioned oscillations, but the available time did not allow further investigations on this control system.

3.4 Chapter Discussion

This chapter has discussed the direct neural network controller. The two implementations show that this controller works.

Since the implemented controllers are single-step ahead controllers, the simulations show typical artefacts of single-step ahead controllers such as damped oscillations of the control signal under certain circumstances.

The direct neural network controller does not need training of a plant model. Instead the controller itself is trained. This is done on simulation runs of the complete system. Hence a careful choice of the training scenario is of importance. Both implementions showed how the final controller was affected by different training scenarios.

The following chapter will summarise all conclusions found throughout this thesis.

Chapter 4
Conclusion

This thesis has investigated two conceptually different approaches to utilise neural networks in control systems. The first one uses the neural network to construct a model of the plant that in turn allows predictive control of non-linear systems. The second uses the neural network in a direct controller where the network output is the control signal.

The following sections present conclusions on both controller types dealt with in this thesis and present some topics of interest for future work.

4.1 Neural Network Model Based Predictive Controller

The generalised predictive control algorithm is extended to non-linear plants by using a neural network based model for prediction.

This concept was then applied to two plants. Methods for system identification were extended to ensure good excitation of the plant during training data generation for plants where standard methods such as a level-change-at-random-instances failed. A combination of random and controlled excitation yielded good results.

The first plant that the controller was tested on is a coupled tank system. It is a non-linear plant with constraints on both the plant states and on the control signal.

The process of creating a neural network based model of the plant – also known as system identification – was solved to a high accuracy. Tests on prediction performance were done to show this.

The neural network model based predictive control algorithm uses these predictions to optimise the projected control series. The simplex optimisation algorithm used here to optimise the projected control series has poor convergence near hard constraints.

This is problematic, since the plant output was constrained to be positive for this plant: The projected control signals did not converge well, and this introduced noise to the control signal.

This showed that the control algorithm is highly dependant on an optimisation algorithm with good convergence.

Even though the poor convergence degraded the performance of the controller, simulation runs with several different kinds of reference signals showed that the control system worked. Apart from the mentioned noise, the controller tracks the reference signal.

The second system is a highly non-linear inverse pendulum. Applying the predictive controller to this system failed for two reasons. This showed some limitations of the discussed methods and highlighted considerations to make before applying them.

Predicting future plant outputs in non-trivial. The predictive control strategy demands a costing horizon of size comparable to the plant dynamics. The prediction of plant output, however, may not be sufficiently accurate over this horizon. This would make the control strategy fail, as it did for the inverse pendulum.

Suggestions were put forward on how to change the model structure to increase the performance of the model by applying a-priori knowledge about the plants kinematic behaviour. Implementation of these suggestions lay outside the scope of this thesis, though.

4.2 Direct Neural Network Controller

The second controller type is a direct neural network controller. This is an output feedback controller where the controller is implemented by a neural network. The implemented controllers are single-step ahead controllers.

Training the neural network is done in a different way for this controller type. In stead of minimising an error function on a training set, the training algorithm minimises the *control* cost function.

This optimisation is done using a simplex optimiser. A simulation run is done to evaluate the cost function for a given neural network. And the result is fed into the optimiser.

The training algorithm was investigated and an algorithm was proposed for finding a start guess for network weights based on a linear model.

First the controller was tested on a reversing trailer truck. The trailer truck is to follow a given path while reversing.

A parameterisation of the path to follow based on Bezier-splines is presented. This allows for an intuitive representation of the path. A path-relative coordinate system known from other literature on path following was applied to the control system. It allows formulation of the control goal as a cost function of plant states.

The control problem was solved in a two step approach, where first a controller is constructed that is able to reverse the truck without the trailer and the truck colliding. Then the controller was extended to make the reversing trailer truck follow the given path.

The resulting controller's ability to control the reversing trailer truck along a new paths was shown, and the sensitivity to change in controller parameters and physical parameters is tested.

The effect of using different training paths for training was also investigated and discussed. When measured on a test path with low curvatures a controller trained on a path with low curvatures was shown to outperform a controller trained on a path with high curvatures.

The direct neural network controller is also tested on the coupled tank system mentioned above.

Due to its different nature, the controller behaves in a different way. The training simulations were chosen to favour a controller that responds quickly to a reference change over a controller that has a low stationary offset. Hence the resulting controller had a noticeable stationary offset.

Furthermore this controller was designed to be a single-step ahead controller, which resulted in a different response to reference changes as compared to the predictive controller.

The control system did not perform as well as the predictive control by means of the deviation of the plant output from the reference. This is caused by the advance that the predictive controller has due to its use of a costing horizon.

Given these facts, the controller performed well if it was not for the hard constraint of the control signal that is not allowed to be negative.

For small control signals, the controller generates oscillating control signals due to the constraint enforced on its output. This situation could be solved by extending the model order which would allow the controller to *learn* that the control signal is limited.

Both controller types require a proper choice of model order, but for the direct neural network controller, the choice of training scenarios is a key factor to obtaining a good controller.

For the neural network model based predictive controller the system identification and the construction of the neural network model of the plant play the key role in controller design.

4.3 Future Work

All in all the implemented control systems show that neural network control can be a feasible solution to a control problem. The suggested controllers can be applied in many cases to obtain better performance on non-linear systems than linear controller can achieve, and in some cases it can control system, that could not be controlled with a linear controller.

But the present thesis also shows that there are many aspects of neural network control that need further investigation.

One topic that has merely been touched upon is model order selection. A step-by-step solution to this problem would be of great help to designers of neural network based control systems.

Another topic is that of noise. Throughout this thesis all signals were assumed to be deterministic. The effects of noise to these systems have not been investigated. Due to the non-linear nature of both the controller and the plants, the extension to non-linear systems of noise theory for linear system is non-trivial.

As for the two controller types investigated here, the more complex model based predictive controller has a potential for good performance on non-linear systems.

But more elaborate work on using neural network models for highly non-linear models would greatly extend the methods reach.

Furthermore work needs to be done on implementing a better optimisation algorithm for the projected control series. Especially the handling of hard constraints on the control signal needs to be handled without severely degrading the convergence of the optimiser.

Appendix A

Matlab Source Code

This appendix lists the sources files written in conjunction with this thesis. The following section A.1 lists the Matlab files used for the neural network model based predictive controller and section A.2 on page 120 lists the Matlab files used for the direct neural network controller.

All files are richly commented, and (almost) all function files contain a header, that explains the arguments passed to the function and the resulting output,

Additionally each of the sections A.1 and A.2 contains a graphical overview of the files in figure A.1 on page 96 and figure A.2 on page 121 respectively.

A.1 Neural Network Model Based Predictive Control

The entire environment used for chapter 2 was coded in Matlab. We will only show the Matlab files for the coupled tank system since the acrobot was done in an almost identical way.

Figure A.1 shows on overview of the Matlab files, that are listed in the following. Note that the file `wb.m` is the main file.

A.1.1 wb.m

```
% Work  bench
%
```

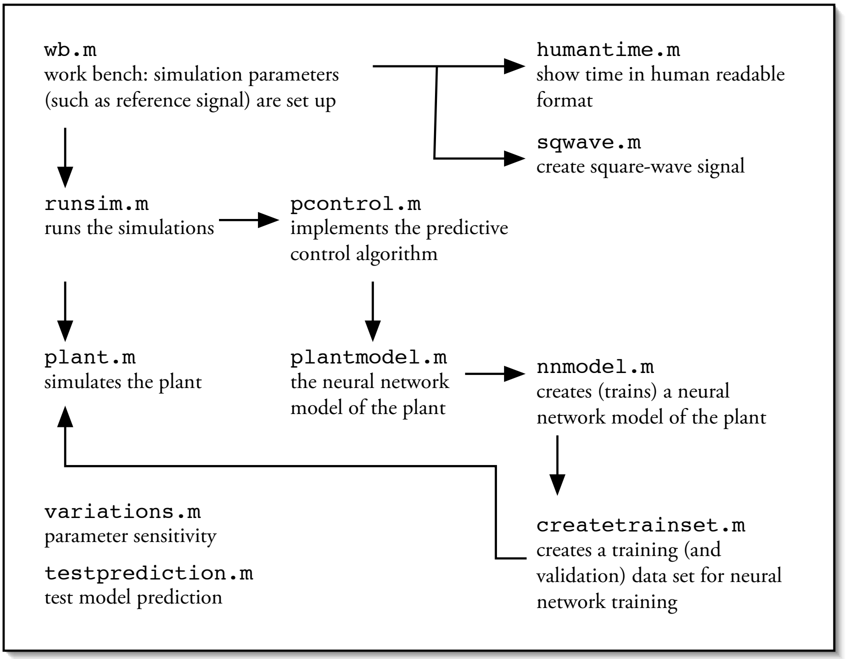

wb.m
work bench: simulation parameters (such as reference signal) are set up
humantime.m
show time in human readable format
sqwave.m
create square-wave signal
runsim.m
runs the simulations
pcontrol.m
implements the predictive control algorithm
plant.m
simulates the plant
plantmodel.m
the neural network model of the plant
nnmodel.m
creates (trains) a neural network model of the plant
variations.m
parameter sensitivity
testprediction.m
test model prediction
createtrainset.m
creates a training (and validation) data set for neural network training

Figure A.1: The Matlab files used for the neural network model based predictive controller. Arrows correspond to function calls.

```
%

Nmax = 800;
U0 = 0;
N2 = 40;
NU = 10;
rho = 100;
R_from = 5;


%
%

global R

switch R_from
  case 1
   R = 0.1*ones(Nmax,1);
  case 2
   R = -0.05*sqwave(Nmax,100)+0.05+0.1;
  case 4
   R = -0.05*sqwave(Nmax,200)+0.05+0.2;
  case 3,
   R(1) = 0;
   for n=2:Nmax,
     if (rand>0.99),
       R(n) = 0.1 + rand*0.30;
     else
       R(n) = R(n-1);
     end
   end
  case 5,
   R = -0.05*sqwave(Nmax,Nmax)+0.05+0.3;
end


runsim( N2, NU, rho, Nmax, U0, 1 );
```

A.1.2 runsim.m

```
function [Y, U] = runsim(N2, NU, rho, Nmax, U_0, doplot, init_arg)
% RUNSIM   Simulate predictive controller system
%
% [Y, U] = RUNSIM( N2, NU, RHO )  Returns the resulting ouput in Y
% and the resulting controlls in U. N2 specifies the predictive
% controller horizon and RHO controls the weight on the control
% signal change. The reference signal has to be present as the
% global variable R.
%
```

```
% [Y, U] = RUNSIM( N2, NU, RHO, NMAX ) Additionally specifies the
% maximum number of iterations to simulate (default = 300).
%
% [Y, U] = RUNSIM( N2, NU, RHO, NMAX, UINIT) Does the same as
% above, but specifies UINIT as the inital controller output
% (defaults to 0).
%
% [Y, U] = RUNSIM( N2, NU, RHO, NMAX, UINIT, DOPLOT ) If DOPLOT is
% positive the resulting controls and output are plotted.
%
% [Y, U] = RUNSIM( N2, NU, RHO, NMAX, UINIT, DOPLOT, ...INITARG )
% Additionally specifies an argument INITARG to be passed to the
% plant initialize function.
%
% The GLOBAL variable Y_POS should be used to limit certain plant
% output variables not to be used.

global R Y_POS

savetraindata = 0;

timeoffset=cputime; % used to measure computation time

if nargin < 4,
  Nmax = 300;
end

if nargin < 7
   plant('init'); % initialize plant model
   plantmodel('init');
else
   plant('init', init_arg); % initialize plant model
   plantmodel('init', init_arg);
end

if nargin < 5
  U_0 = zeros( plant('inputcount'), 1);
end

if nargin < 6
   doplot = 0;
end
if doplot > 1
   doderiv = 1; % calculate (and plot) derivates
else
   doderiv = 0;
end
```

```
% setup simulation
U(1,:) = U_0;
Y(1,:) = plant('actuate', U_0);
%plantmodel('actuate', U_0); %%% DEBUG%%%
 if Y_POS == [],
   Y_POS = 1:size(Y,2);
   clear_y_pos = 1;
 else
   clear_y_pos = 0;
 end

% pad reference
R2 = [ R; R(end)*ones(N2,1) ];

% run simulation
n = 2;
nCount=floor(Nmax/100);
 if nCount == 0, nCount = 1; end
 for n=1:Nmax
   if mod(n,nCount) == 0,
     fprintf('simulation with %i steps: %3i%% done\r', ...
                 Nmax, floor((n-1)/Nmax*100));
   end
   if 0,
     U_n = 0.5*randn;
   else
     U_n = pcontrol( NU, n-1, R2(n:n+N2-1), rho );
   end
   U(n,:)=U_n;
   Y(n,:)=plant('actuate', U(n));
   %Y(n,:)=plantmodel('actuate', U(n));  %%%DEBUG%%%
   if (savetraindata > 0),
     % extract state info
     state = plant('savestate');
     X(n,:) = state.X';
   end
 end
N = n - 1;
 fprintf('\r                                      \r');
 if (clear_y_pos == 1)
   clear global Y_POS;          % Clean up Y_POS
 end

 fprintf( [ 'CPU time used: ' ...
           humantime( cputime-timeoffset ) '\n' ]);

 if ( doplot > 0),
   nplots=2;
   newplot;
```

```
    % number of simulation step to plot
    N_max = Inf;
    N_max = min(N_max, N); % limit to N
    n=1:N_max;
    yrow = 1; % row in y-matirx to plot

    subplot(nplots,1,1);
    if (savetraindata == 0),
      plot(n,Y(1:N_max,yrow), n,R(1:N_max));
      legend('y','r',0);
    else
      plot(n,X(1:N_max,1), n,X(1:N_max,2), n,R(1:N_max));
      legend('x1','x2','r',0);
    end

    subplot(nplots,1,2);
    plot(n,U(1:N_max));
    legend('u',0);

    drawnow;
  end

  if (savetraindata > 0),
    U = U';
    Y = X';
    save traindata U Y
  end
```

A.1.3 plant.m

```
function result=plant( action, input)
% PLANT  Simulate plant
%
% RESULT = PLANT( ACTION, INPUT) Simulates the plant. Action
% specifies the action to be taken and can be one of the following:
% o 'init' Initialise plant. The must be called before using
%   'measure' or 'actuate'. No input nor return value.
% o 'savestate' Returns the current state in RESULT.
% o 'restorestate' Restores the state from the value passed in
%   INPUT, which in turn should be obtained by an (earlier) call to
%   'savestate'
% o 'measure' Returns in RESULT the current measurement. This does
%   not affect the plant's state.
% o 'acutate' Takes INPUT as the control signal(s) to the plant and
%   acutates the plant. Plant states will be updated accordingly
%   and the plant output will be returned in RESULT.
% o 'outputcount' / 'inputcount' return in RESULT the number of
%   output and input signals respectively.
% o 'plot' (if defined) will take plant specific data as INPUT and
%   plot it.
%
% When simulating the plant, first call 'init' and then use
% 'actuate'. 'savestate' and 'restorestate' (if implemented) allow
% simulation to go back in time, if neccessary.
%
% Several global variables are maintained, all with names starting
% with 'plant'. They should be considered private to the function
% and be left unaltered.
%
% This implementation simulates a coupled tank system.
%


global plantX plantA plant1C plantTankheight
global plantTs plantU
global plantOptions
global plantStoreX plantStoreU

switch lower( action )
 case 'init'
  %fprintf('Coupled tank system\n');

  if nargin > 1,
    scenario = input * 1;
  else
    scenario = 0;
  end

  % set up variables
  % units are cm, cm^2, etc
  plantX = [ 0.10; 0 ];

  C = 0.0153928; % tank footprint
  g = 9.81; % gravity
  ao = 0.00005; al = 0.00008; % pipe footprint
  so = 0.44; sl = 0.31;
  a1 = so*ao*sqrt(2*g)/C;
  a2 = sl*al*sqrt(2*g)/C;

  global VARIATE
  if ( size( VARIATE, 1 ) > 0 ),
    switch VARIATE.parameter
     case 1
      ao = ao * VARIATE.change;
     case 2
      al = al * VARIATE.change;
    end
```

```
   end

   plantA = [ -a1 a2; a1 -2*a2 ];
   plant1C = [0;1/C];
   plantTs = 5; % sampling period

   plantTankheight = 0.6;

  % set up options for ODE solver
   plantOptions = odeset('MaxOrder', 3, ...
                         'RelTol', 1e-7, ...
                         'AbsTol', [1e-7 1e-7], ...
                         'Refine', 1);

  % variables to store plant input / output
  % (for post mortem plotting )
   plantStoreX = [];
   plantStoreU = [];

 case 'savestate'
  % save states
   result.X = plantX;

  % save lag space for plant model
   global plantmodelLagU plantmodelLagY
   for n=1:size(plantmodelLagU,1)
     if (n > size(plantStoreU,1))
       result.U(n,:) = 0;
     else
       result.U(n,:) = plantStoreU(end+1-n);
     end
   end
% $$$    for n=1:size(plantmodelLagY,1)
% $$$      if (n > size(plantStoreX,1))
% $$$        result.Y(n,:) = 0;
% $$$      else
% $$$        result.Y(n,:) = plantStoreX(end+1-n,1);
% $$$      end
% $$$    end

 case 'restorestate'
   plantX = input.X;

 case 'measure'
   result = plantX(1);

 case 'actuate'
   n = size(plantStoreU,1);
  % get old control
  if n == 0,
    plantU = 0;
  else
    plantU = plantStoreU( n, : );
  end

  % we cannot pump water out of tank 1
  if (plantU < 0),
    plantU = 0;
  end

  % solve differential equations
  y0 = plantX;
  [T,Y] = ode45( @plantDiffM, [0 plantTs], y0, plantOptions  );
  % update states
  plantX = Y(end,:)';

  % check for tank levels below 0:
  if plantX(1) < 0,
    plantX(1) = 0;
  end
  if (plantX(2) + plantX(1)) < 0,
    plantX(2) = - plantX(1);
  end
  % ... and check for tanks levels above maximum
  if (plantX(1) > plantTankheight),
    plantX(1) = plantTankheight;
  end
  if ( (plantX(2) + plantX(1)) > plantTankheight );
    plantX(2) = plantTankheight - plantX(1);
  end

  % return result
  result = plantX(1);

  if result > 0.6,
    message = sprintf('Problem with states X1 = %f > 0.6!', ...
                      result);
    warning(message);
  end

  % store control signal (input) and outputs
  plantStoreU(n+1,:) = input;
  plantStoreX(n+1,:) = plantX';

 case 'outputcount'
  result = 1;
```

```
   case 'inputcount'
    result = 1;

   case 'plot'
    subplot(2,1,1);

end

% sub-functions

function dy = plantDiffM( t, y );
global plantX plantA plant1C plantU
dy = plantA*(sqrt(abs(y)).*sign(y))+plant1C*plantU;
```

A.1.4 pcontrol.m

```
function u = pcontrol( NU, t, r, rho )
% CONTROL  Calculate control signal
%
% U = PCONTROL( T, R ) returns in U the output of the non-linear
% predictive controller for time t=T. R contains the reference
% signal, such that R(1,:) is the reference for t=T and R(2,:) is
% the reference for t=T+Ts.
%
% If T > 0 then PCONTROL requires that PCONTROL( T-1, ...) has been
% called previously.
%
% PCONTROL uses PLANTMODEL for predicting the plant outputs.
%
% U = PCONTROL( T, R, RHO ) Weights the control signal change with
% rho.
%
% Uses several global variables (names starting with CONTROL_).

% DU corresponds to \Delta U = U - q^{-1} U
% U corresponds to U

global CONTROL_rho CONTROL_ref CONTROL_U CONTROL_DU
global CONTROL_offsetU;
global CONTROL_state CONTROL_stateModel
global CONTROL_usemodel

CONTROL_usemodel = 1;
% set to 0 for DEBUG (uses real model for prediction)


t = t + 1; % array offset
quiet = 1; % printing of iterations
if nargin < 3,
  CONTROL_rho = 0;
else
  CONTROL_rho = rho;
end

% save time using global variable
CONTROL_ref = r;
N2 = size(r,1);

% Get the lag network from the 'real' plant. 'savestate' will also
% retreive the state information, but the PLANTMODEL will not use
% this information, but only the lag network of old input /
% ouput. This way we have a (rather) clean way of moving data from
% the plant to the plant model.
CONTROL_state = plant('savestate');
% Get the info from the model, so we can restore its state after
% predicting:
CONTROL_stateModel = plantmodel('savestate');


if quiet,
  options = optimset('Display','none');
else
  options = optimset('Display','iter');
end
%options = optimset(options,'TolX',1e-8);
options = optimset(options,'LargeScale','off');

% get start guess if exists and set maximum iterations accordingly
if (t-1) == 0, % we've shifted t to t+1
  CONTROL_offsetU = 0;
  DU = 0.00001*randn(NU,1);
  iterations = 16;
  CONTROL_DU = [];
else
  CONTROL_offsetU = CONTROL_U(t-1);
  DU = [CONTROL_DU(t:(t+NU-2),:); 0];
  iterations = 8;
end

options = optimset(options,'MaxIter',4*NU);

iteration = 0;
exitflag = 0;
while ((iteration < iterations) & (exitflag == 0)),
  iteration = iteration + 1;

  % find control signals
  [DU, fval, exitflag] = fminsearch(@costfunction,DU,options);
```

```
    % limit signals > 0
     % calculate future absolute control signals
     U = CONTROL_offsetU + DU(1);
     for n=2:size(DU,1),
       U(n)=U(n-1)+DU(n);
     end

     if size(find(U<0)),
       exitflag = 0;
     end

     U = U .* ( U > 0 );
     % calculate control signal increase
     DU(1) = U(1) - CONTROL_offsetU;
     for n=2:size(DU,1),
       DU(n) = U(n)-U(n-1);
     end
end

% store signals (to use as start guess at next function call
CONTROL_DU(t:(t+NU-1),:)=DU;
CONTROL_U(t)=CONTROL_offsetU+CONTROL_DU(t);

% restore plant state
plantmodel('restorestate',CONTROL_stateModel);
plant('restorestate',CONTROL_state);

% return value
u = CONTROL_U(t);



function J = costfunction( DU )
% calculate the cost function when controlling with control
% signals given by U
global CONTROL_rho CONTROL_ref CONTROL_state CONTROL_offsetU
global CONTROL_usemodel
N2 = size(CONTROL_ref,1);
NU = size(DU,1);
% Put state into plant
if (CONTROL_usemodel > 0),
   plantmodel('restorestate', CONTROL_state);
else
   plant('restorestate',CONTROL_state);
end
% run simulation
U = CONTROL_offsetU;
```

```
for n=1:N2,
  if ( n ≤ NU ),
    U = U + DU(n);
  end
  if (CONTROL_usemodel > 0),
    Y(n,:)=plantmodel('actuate', U);
  else
    Y(n,:)=plant('actuate', U);
  end
end
% evaluate cost function
J = sum((CONTROL_ref-Y).^2)/N2 + ...
    CONTROL_rho*sum(DU.^2)/NU;
```

A.1.5 plantmodel.m

```
function result=plantmodel( action, input)
%PLANT  Simulate plant (model)
%
%  RESULT = PLANTMODEL( ACTION, INPUT) Runs the plant model. This
%  is very much like PLANT with the difference, that PLANTMODEL is
%  the model known to the controller, while PLANT is the simulation
%  of the actual plant.
%
%  When using the plant model, first call 'init' and then use
%  'actuate'.
%
%  Several global variables are maintained, all with names starting
%  with 'plantmodel'. They should be considered private to the
%  function and be left unaltered.
%
%  The plant model is created as a neural network model of the
%  actual plant implemented in PLANT. Calling
%       plantmodel( 'init', 'retrain' )
%  will cause the neural network plant model to be retrained by
%  calling NNMODEL. Otherwise
%       plantmodel( 'init' )
%  will try to load the model from a file called 'plantnnmodel.mat'
%  to save time.
%
%  See also PLANT, NNMODEL


global plantmodelNet plantmodelW1 plantmodelW2
global plantmodelLagU plantmodelX
global plantmodelMeanP plantmodelStdP
global plantmodelMeanT plantmodelStdT

switch lower( action )
```

```
case 'init'
 % clear existing plantmodelNet
 plantmodelLagU == [];

 retrain = 0;
 if nargin > 1,
   if input == 'retrain',
     retrain = 1;
   end
 end

 if ~retrain ,
   % try to load from file
   try
     load plantnnmodel;
   end
   if plantmodelLagU == [],
     retrain = 1;
   end
 end

 % lag space
 plantmodelLagU = zeros( 3+1 , 1 ); % number of values to lag
                                    % plus current
 % plant model state
 plantmodelX = zeros(2 ,1);

 % if we still have no nn model retrain one
 if retrain ,
   % nn model
   plantmodelNet = nnmodel( size( plantmodelLagU , 1) - 1 , ...
                            0 , ...
                            12, 40000 );
   % save it to a file
   save plantnnmodel plantmodelNet plantmodelLagU plantmodelX
 end

 % reset lag space ( in case we loaded it )
 % ... you never know

 plantmodelLagU = 0 * plantmodelLagU;

 % To speed up things , we'll use our own implementation of
 % network/sim . To do this we extract the weight and bias info:

 plantmodelW1 = [ plantmodelNet.b{1 ,1} plantmodelNet.IW{1 ,1} ];
 plantmodelW2 = [ plantmodelNet.b{2 ,1} plantmodelNet.LW{2 ,1} ];

 plantmodelMeanP = plantmodelNet.userdata.meanp;
```

```
     plantmodelStdP = plantmodelNet.userdata.stdp;
     plantmodelMeanT = plantmodelNet.userdata.meant;
     plantmodelStdT = plantmodelNet.userdata.stdt;

    case 'savestate'
     result.U = plantmodelLagU;
     result.X = plantmodelX;

    case 'restorestate'
     plantmodelLagU = input.U;
     plantmodelX = input.X;

    case 'measure'
     result = plantmodelX(1);

    case 'actuate'
     % simulate plant

     % get neural network input
     p = [plantmodelLagU; plantmodelX];

     % calculate output
     %p = (p-plantmodelMeanP)./plantmodelStdP;
     p = trastd( p, plantmodelMeanP, plantmodelStdP );
     y = plantmodelW2 * [ 1; tanh( plantmodelW1 * [ 1; p ] ) ];
     %y = y .* plantmodelStdT + plantmodelMeanT;
     y = poststd( y, plantmodelMeanT, plantmodelStdT );

     % shift control input output into lag space
     plantmodelLagU = [ input; plantmodelLagU(1:end-1) ];

     % update plant model state
     plantmodelX = y;

     result = plantmodelX(1);

    case 'outputcount'
     result = 1;

    case 'inputcount'
     result = 1;

    case 'plot'
     warning('''plot''␣is␣not␣implemented');

    case 'plottrain'
     tr = plantmodelNet.userdata.tr;
     subplot(1,1,1);
     semilogy(tr.epoch, tr.perf, 'k', ...
        tr.epoch, tr.vperf, 'b', ...
        tr.epoch, tr.tperf, 'r', ...
            'LineWidth', 2)
  axis auto
  xlabel('epochs');
  ylabel('error function (black: training  blue: validation)');
  grid on;
end
```

A.1.6 nnmodel.m

```
function [net, tp] = nnmodel( lagu, lagy, hidden, N, verbose ),
%NNMODEL   Create NN model for plant
%
%  NET = NNMODEL( LAGU, LAGY ) Creates a neural network model of
%  the plant defined by PLANT. The parameters LAGU and LAGY specify
%  how many control inputs and plant outputs respectively are to be
%  contained in the lag network for the plant.
%
%  E.g. for LAGU = 0 and LAGY = 1 only the current control signal
%  used as input along with the corrent and previous plant output.
%
%  The network is created using NEWFF and trained using TRAIN.
%
%  NET = NNMODEL( LAGU, LAGY, N ) N additionally specifies the
%  number of samples to create in the training set for the neural
%  network. N defaults to 3000.
%
%  See also PLANT, NEWFF, TRAIN

usetestset = 0;

% number of samples
if nargin < 4,
  N = 10000;
end
% verbose
if nargin < 5,
  verbose = 1;
end

global NNMODEL_P NNMODEL_T NNMODEL_VP NNMODEL_VT
global NNMODEL_TP NNMODEL_TT

if size(NNMODEL_T,2) == N
  P = NNMODEL_P;
  T = NNMODEL_T;
  VP = NNMODEL_VP;
  VT = NNMODEL_VT;
```

```
    if usetestset,
      TP = NNMODEL_TP;
      TT = NNMODEL_TT;
    end
  else
    % training set
    [P,T]=dataset(lagu,lagy,N,1);                % training set
    % validation set
    [VP,VT]=dataset(lagu,lagy,floor(N/2),2); % validation set
    if usetestset,
      % test set
      [TP,TT]=dataset(lagu,lagy,floor(N/2),3); % test set
    end

    % store data for next time
    NNMODEL_P  = P;
    NNMODEL_T  = T;
    NNMODEL_VP = VP;
    NNMODEL_VT = VT;
    if usetestset,
      NNMODEL_TP = TP;
      NNMODEL_TT = TT;
    end
  end

  % do pre-processing of data:
  % Normalise the mean and standard deviation
  [P, meanp, stdp, T, meant, stdt] = prestd( P, T );
  VP = trastd( VP, meanp, stdp );
  VT = trastd( VT, meant, stdt );
  if usetestset,
    TP = trastd( TP, meanp, stdp );
    TT = trastd( TT, meant, stdt );
  else
    TP = [];
    TT = [];
  end

  % setup NN
  net = newff( minmax( P ), [hidden, size(T,1)], ...
              {'tansig','purelin'}, 'trainlm');
  net.trainParam.show = NaN;
  net.trainParam.mu_max = 1e12;
  net.trainParam.time = Inf;
  net.trainParam.max_fail = 25;

  % validation data
  VV.P = VP;
  VV.T = VT;
```

```
VV.Pi = [];
VV.Ai = [];
% test data
if usetestset,
  TV.P = TP;
  TV.T = TT;
  TV.Pi = [];
  TV.Ai = [];
else
  TV = [];
end

oldwarn = warning('off');

% select best NN out of a small series
net.trainParam.epochs = 50;
net.trainParam.goal = 1e-8;
tries = 6;
bestnet = [];
besttr.vperf=Inf;
for i=1:tries
  fprintf( 'Initial␣cross-validation␣training␣#␣%i␣of␣%i\r', ...
           i, tries );
  % setup weights
  net = init(net);
  % train to 1st goal without validation data
  [net,tr]=train(net,P,T,[],[],VV,[]);
  if tr.vperf(end) < besttr.vperf(end),
    % this one is better
    bestnet = net;
    besttr = tr;
  end
end
net = bestnet; bestnet = [];
fprintf( '␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\r' );

if verbose,
  net.trainParam.show = 50;
end

% train to final goal with validation data
net.trainParam.epochs = 1200;
net.trainParam.goal = 1e-8; % final goal
[net,tr]=train(net,P,T,[],[],VV,TV);

warning(oldwarn);

% Post-training analysis
% (not done; see postreg)
% put pre-poroccessing data into neural network userdata:
userdata.meanp = meanp;
userdata.stdp = stdp;
userdata.meant = meant;
userdata.stdt = stdt;
% and put training record into userdata
userdata.tr = tr;

net.userdata = userdata;

if nargout > 1
  tp = tr.tperf(end);
end




function [P, T]=dataset(lagu, lagy, N, setselect),
% Get dataset of controls and plant outputs

if lagy > 0,
  error('lag_y_must_be_0');
  % using model state instead of lag space for y
end

switch setselect,
 case 1,
  name = 'training';
 case 2,
  name = 'validation';
 case 3,
  name = 'test';
 otherwise,
  name = 'unknown';
end

if N > 0,
  % initialise plant
  plant('init');

  U=[0];
  % get initial state
  Y = plant('measure');
  state = plant('savestate');
  X = state.X;

  % create control signal and get plant response
  nCount=floor(N/100);
  if nCount == 0, nCount = 1; end
  stop = 0;
  for n=2:N,
    if mod(n,nCount) == 0,
      fprintf( ['Creating␣' name ...
                '␣set␣with␣%i␣samples:␣%3i%%␣done\r'], ...
               N, floor((n-1)/N*100) );
    end

    if ( Y(1,n-1) > 0.55 ),
      poscount = poscount + 1;
    else
      poscount = 0;
    end
    if (poscount > 25) & (stop < 1),
      stop = floor(rand*100);
     U(n) = 0;
    end
    if ( U(n-1) < -0.00005 ) & ( Y(1,n-1) < 0.05 ),
      negcount = negcount + 1;
    else
      negcount = 0;
    end

    stop = stop - 1;
    if (stop < 1),
      if rand > 0.95
        U(n) = U(n-1)+2*(rand-0.52)*0.0010;
        a(n) = 1;
      else
        U(n) = U(n-1);
        a(n) = 0;
      end
     U(n) = U(n)+(rand-0.5)*0.0002;
    else
     a(n) = -1;
     U(n) = rand*0.00005;
    end
    if negcount > 60,
     a(n) = -2;
     U(n) = rand*0.0001;
    end
%    if U(n) < 0,
%      U(n) = U(n) * 0.1;
%    end
    % actuate plant and get output
    Y(:,n) = plant('actuate', U(n));
    state = plant('savestate');
    X(:,n) = state.X;
  end

  fprintf(['␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣' ...
           '␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\r']);
else
  % try to load from file
  load traindata
  N = size(U,2);
  % Use 1st half for training, and divide 2nd half into validation
  % and test set:
  switch setselect,
   case 1
    U = U(:,1:floor(N/2));
    Y = Y(:,1:floor(N/2));
   case 2
    U = U(:,floor(N/2)+1:floor(3*N/4));
    Y = Y(:,floor(N/2)+1:floor(3*N/4));
   case 3
    U = U(:,floor(3*N/4)+1:N);
    Y = Y(:,floor(3*N/4)+1:N);
  end
  % Get new size
  N = size(U,2);
end

if 1,
  % plot
  subplot(3,1,1);
  plot(X');
  title(sprintf('%s␣set␣with␣%i␣samples', name, N));
  legend('y','X2',0);
  subplot(3,1,2);
  plot(U');
  legend('u',0);
  subplot(3,1,3);
  plot( a', '.');
  legend('a',0);
  drawnow;
end

% Reshape input / output data
% to a lag-network of input data
P = []; T = [];
for n=1:lagu+1,
  P(n, :) = [ zeros(1,n) U(1:N-n) ]; % lag controls
end
if 0,
  % lag outputs
  for n=1:lagy+1,
    P(n+lagu+1, :) = [ ones(1,n)*Y(1) Y(1:N-n) ]; % lag outputs
  end

  % Neural network target values are the outputs:
  T = Y;
else
  % use states
  P(lagu+2:lagu+3, :) = [X(:,1) X(:,1:end-1)];
  % Neural network target values are the states
  T = X;
end

% Output data set size
name(1)=upper(name(1));
fprintf( [ name '␣set␣has␣%i␣samples.\n'], N );
```

A.1.7 createtrainset.m

```
%
% Create training data set
%
% for system model
%

% number of samples
N = 1000;

U=[];
Y=[];

% create input data

maxu = 0.2;
minu = -0.2;
alpha = 1-1/10;

fprintf( 'Creating␣input␣signal␣with␣range␣%f␣to␣%f\r', ...
         minu, maxu );

fprintf( ['␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣' ...
          '␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\r'] );

for n=1:N,
  if (n == 1) | (rand > alpha),
    U(n) = minu + (maxu-minu)*rand;
  else
    U(n) = U(n-1);
  end
```

```
end

% initialise plant
plant('init');

for n=1:N,
  fprintf( 'Simulating␣plant␣with␣%i␣samples:␣%3i%%␣done\r' , ...
           N, floor((n-1)/(N-1)*100) );
  Y(n) = plant('actuate',U(n));
end

fprintf( ['␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣' ...
          '␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\r'] );

% plot
subplot(2,1,1);
plot(Y);
legend('y',-1);
subplot(2,1,2);
plot(U);
legend('u',-1);
drawnow;

fprintf( 'Data␣set␣ready␣in␣U␣and␣Y.\n' );

P = []; T = [];
lagu = 2;
for n=1:lagu+1,
  P(n, :) = [ zeros(1,n-1) U(1:N-n+1) ];
end
lagy = 2;
for n=1:lagy+1,
  P(n+lagu+1 , :) = [ ones(1,n-1)*Y(1) Y(1:N-n+1) ];
end
```

A.1.8 sqwave.m

```
function u=sqwave(n,per)
% SQWAVE   Create square wave signal
%
% U = SQWAVE( N, PER ) Creates a square wave of length N with a
% period eqaul to PER.

% Programmed 2002 by Daniel Eggert
% Department of Mathematical Modelling,
% Technical University of Denmark

m = rem((1:n)*(1/per),1);
```

```
m = (m+(m==0))≤0.5;
u = (2*m-1)';
```

A.1.9 humantime.m

```
function S = humantime( s ),

m = floor(s/60);
s = mod(s,60);

h = floor(m/60);
m = rem(m,60);

if h > 0,
  S = [ int2str(h) 'h␣' ];
else
  S = [];
end
if (h > 0) | (m > 0),
  S = [ S int2str(m) '''␣' ];
end
S = [ S int2str(s) '''''␣' ];
```

A.1.10 variations.m

```
%
% Run through parameter variations and plot resulting costfunction
%
% Will variate physical parameters defined in plant.m by using
% global variable



global R
Nmax = 800;
U0 = 0;
N2 = 40;
NU = 10;
rho = 100;
R = -0.05*sqwave(Nmax,200)+0.05+0.2;


fprintf(['Computing␣cost␣function␣for␣physical␣parameter␣' ...
         'variations:␣\n']);


% array of parameters
parameter=1:2;
parameter_idx=1:size(parameter,2);
```

```

% array of relative changes
rel_change=0.9:0.002:1.1;
rel_change_idx=1:size(rel_change,2);

fprintf('Will do a total of %i simulations runs...\n', ...
        parameter_idx(end)*rel_change_idx(end)+1 );

% matrix for results
J_var = zeros( parameter_idx(end), rel_change_idx(end) );

% set up VARIATE variable for modifying physical parameters
% in plant.m
global VARIATE
VARIATE = [];

% Get costfunction for original parameters
[Y,U] = runsim( N2, rho, Nmax, 0, 1 );
DU = U(1:end-1)-U(2:end);
J_org = sum((R-Y).^2)/size(R,1) + ...
        rho*sum(DU.^2)/size(DU,1);

timeoffset=cputime;
for i=parameter_idx,
  for change=rel_change_idx,
    fprintf('\r%i%% ', ...
            floor(100*((i-1)*rel_change_idx(end)+change-1)/ ...
                  (rel_change_idx(end)*parameter_idx(end)) ) );

    % change the parameters
    VARIATE.parameter = parameter(i);
    VARIATE.change = rel_change(change);

    % calculate costfunction for these weights
    [Y,U] = runsim( N2, NU, rho, Nmax, 0, 1 );
    DU = U(1:end-1)-U(2:end);
    J = sum((R-Y).^2)/size(R,1) + ...
        rho*sum(DU.^2)/size(DU,1);

    % store costfunction result
    J_var( i, change ) = J;
    save variations.mat J_org J_var rel_change
  end
end
fprintf( ['CPU time used: ' humantime( cputime-timeoffset ) '\n' ]);

% normalise costfunctions with original ostfunction
J_varnorm = J_var / J_org;
```

```
% plot results
plot(rel_change ,J_varnorm)
axis( [ rel_change(1) rel_change(end) 1 2 ] );
grid on;
xlabel('relative change of parameter');
ylabel('relative chaneg in cost funxtion');
legend('l_1','l_2','v_2',0);

% save results
fprintf('Saving results to file ''variations.mat''\n');
save variations.mat J_org J_var J_varnorm rel_change
```

A.2 Direct Neural Network Control

Below the Matlab files used for chapter 3 are listed. Again only files for one of the two implementations are listed.

The reversing trailer truck is slightly more complex as it includes files to implement the Bezier-spline based path. An overview of its files are shown in figure A.2.

A.2.1 wb.m

```
%
% This is the workbench
%
% Run FMINSEARCH to find neural network weights
%

global RUNSIM_BATCH_COUNT Y_POS

RUNSIM_BATCH_COUNT = 4; % Number of different initial scenarios
weightsFrom=0;  % Where to get the inital weights from (0 = load
                % from file)
search = 0;     % If set to 0 the weights will remain unchanged
saveWeights=search; % Whether to save resulting weights to file
plotmotion=0;   % Whether to plot the motion of the vehile
maxIter = 100;  % Number of iterations to run with fminsearch
%Nmax = 100;      % max number of simulation steps ( N*Ts )
plant('init');Nmax = bezierlength*3;
resetR = 1;     % get new references
rho = 0.001;     % control cost weight
quiet = 0;


switch weightsFrom
```

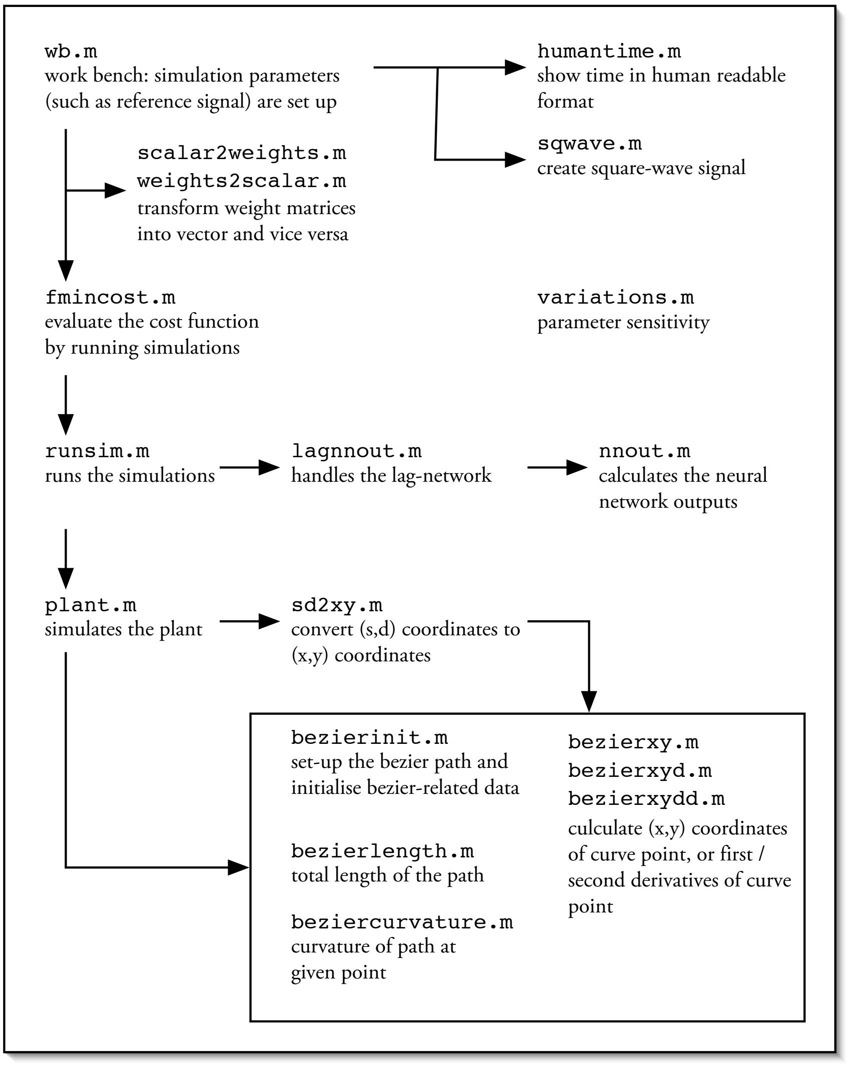

wb.m
work bench: simulation parameters (such as reference signal) are set up
humantime.m
show time in human readable format
sqwave.m
create square-wave signal
scalar2weights.m
weights2scalar.m
transform weight matrices into vector and vice versa
fmincost.m
evaluate the cost function by running simulations
variations.m
parameter sensitivity
runsim.m
runs the simulations
lagnnout.m
handles the lag-network
nnout.m
calculates the neural network outputs
plant.m
simulates the plant
sd2xy.m
convert (s,d) coordinates to (x,y) coordinates
bezierinit.m
set-up the bezier path and initialise bezier-related data
bezierlength.m
total length of the path
beziercurvature.m
curvature of path at given point
bezierxy.m
bezierxyd.m
bezierxydd.m
culculate (x,y) coordinates of curve point, or first / second derivatives of curve point

Figure A.2: This is how the Matlab files for the direct neural network controller fit together. Arrows correspond to function calls.

```
    case 0
     load weights.mat
    case 1
     % zero all 1st level weights and unit all 2nd level weights
     n_R = 0;
     n_Y = 1;
     n_U = 1;
     n_hidden = 1;
     % Y = [s d theta alpha_1 alpha_2 curv(s)]
     Y_POS = [ 2:6 ];
     size_U = plant( 'inputcount' );
     size_Y = size( Y_POS, 2 );
     % zero weights
     W1 = zeros( n_hidden, 1 + n_R*size_Y + n_U*size_U + n_Y*size_Y );
     % W1 /=/ [offset d theta alpha_1 alpha_2 curv(s) U]
     %W1(1,2)=-0.1;
     global glP
     W1(1,3) = glP(1)/10; % theta
     W1(1,4) = glP(2)/10; % alpha_1
     W1(1,5) = glP(3)/10; % alpha_2
     W2 = [0 ones( size_U, (n_hidden) )]*10;
    case 2 % result of finda
     n_R = 0;
     n_Y = 1;
     n_U = 1;
     n_hidden = 1;
     Y_POS = [ 2:6 ];
     W1 = zeros( 1, 1 + 0 + 1 + 5 );
     W1(1,3) = -1.3635/10;
     W1(1,4) = 4.0756/10;
     W1(1,5) = -3.3265/10;
     W2 = [0 10];
    case 3
     load weights.mat
     W1 = W1/2;1
     W2 = [W2(:,1) W2(:,2:end)*2];
    case 10
     % go from using offset to not using offset (scalar2weights)
     load weights.mat
     W1(:,1) = 0*W1(:,1);
     W2(:,1) = 0*W2(:,1);
    case 20
     n_hidden = 1;
     Y_POS = [ 2:6 ];
     size_U = plant( 'inputcount' );
     size_Y = size( Y_POS, 2 );
     n_R=0;
     n_U=1;
     n_Y=1;
```

```
      load weights3.mat
    otherwise
      % neural network structure
      n_R = 0; % number of references in lag network
      n_Y = 1; % number of plant outputs --''--
      n_U = 1; % number of controls       --''--
      n_hidden = 1; % number of hidden units in neural network
      Y_POS = [ 2:6 ];
      size_U = plant( 'inputcount' );
      size_Y = size( Y_POS, 2 );
      % random weights
      W1 = randn( n_hidden, 1 + n_R*size_Y + n_U*size_U + n_Y*size_Y );
      W2 = randn( size_U, (n_hidden+1) );
  end

  global R
  if resetR,
    % force new reference function
    R = [];
  end

  % Search for minimum using fminsearch
  [SW, S1, S2] = weights2scalar( W1, W2, 1 );
  if search,
    options = optimset( 'Display', 'iter', 'MaxIter', maxIter);
    timeoffset=cputime;
    [SW, Jmin, exitflag] = fminsearch( @fmincost, SW, options, S1, ...
                                       S2, n_R, n_U, n_Y, rho, ...
                                       Nmax, -1);
    fprintf( ['CPU time used: ' ...
              humantime( cputime-timeoffset ) '\n' ]);
  end
  [W1, W2] = scalar2weights( SW, S1, S2, 1 );

  if ~quiet,
    % output result
    fprintf('Resulting neural network weights:\n');
    W1,W2
  end

  if saveWeights == 1,
    save weights.mat W1 W2 n_R n_Y n_U Y_POS
  end

  % Run simulation with found weights (and plot)
  [Jmin,Y,U] = fmincost( SW, S1, S2, n_R, n_U, n_Y, rho, Nmax, ...
                         ~quiet);
  if ~quiet,
    fprintf('J = %f\n', Jmin);
```

```
end

if plotmotion == 1,
  %pause
  plant( 'plot' );
end
```

A.2.2 scalar2weights.m

```
function [W1, W2] = scalar2weights(X, S1, S2, nooffset)
% SCALAR2WEIGHTS  Create weight matrices from scalar vector
%
% [W1, W2] = SCALAR2WEIGHTS(X, S1, S2) The content of X and the
% sizes S1 and S2 are used to create the weight matrices W1 and
% W2.

if nargin < 4,
  nooffset = 0;
end
if nooffset,
  S1(2) = S1(2) -1;
  S2(2) = S2(2) -1;
end

for n=1:S1(1)
  W1(n,:)=X( 1+(n-1)*S1(2):n*S1(2) );
end
if nooffset,
  W1 = [ zeros(S1(1),1) W1 ];
end
X = X(1+S1(1)*S1(2):end);
for n=1:S2(1)
  W2(n,:)=X( 1+(n-1)*S2(2):n*S2(2) );
end
if nooffset,
  W2 = [ zeros(S2(1),1) W2 ];
end
```

A.2.3 weights2scalar.m

```
function [X, varargout] = weights2scalar(W1, W2, nooffset)
% WEIGHTS2SCALAR  Put containt of weight matrices into scalar
%
% [X] = WEIGHTS2SCALAR(W1, W2) puts all the elemtns of W1 and W2
% into the scalar X.
% [X, S1, S2] = WEIGHTS2SCALAR(W1, W2) also outputs the size of W1
% and W2 into S1 and S2 respectively.

if nargin < 3,
  nooffset = 0;
end
if nooffset,
  start = 2;
else
  start = 1;
end

X=[];
for n=1:size(W1,1)
  X = [X W1(n,start:end)];
end
for n=1:size(W2,1)
  X = [X W2(n,start:end)];
end
if nargout > 1
  varargout(1) = {size(W1)};
  varargout(2) = {size(W2)};
end
```

A.2.4 fmincost.m

```
function [J,Y,U] = fmincost( scalar, S1, S2, n_R, n_U, n_Y, rho, ...
                              Nmax, doplot )
% FMINCOST   Calculate cost function for nn control
%
% J = FMINCOST( SW, S1, S2, NR, NU, NY, RHO )
% J = FMINCOST( SW, S1, S2, NR, NU, NY, RHO, N, DOPLOT )
% Calculates the costfunction for a neural network with a lag
% network specified by NR, NU and NY. These are the number of
% references, control and output signals contained in the lag
% network feeding the input to a neural network.
%
% This function can be used as a cost function for fminsearch.
%
% The neulral network structure and weights are specified by SW,
% S1, S2. The weights are extracted by using SCALAR2WEIGHTS.
%
% Nmax spedifies the maximum number of samples to simulate. It is
% optional and defaults to 100.
%
% The parameter DOPLOT is optional. If specified and set to 1 the
% simulation results will be plotted. If set to 0 no plotting will
% be done (default) and if set to -1 a plot will be done every
% 30th function call.
%
% The reference signal is a square wave function, and the
% costfunction evaulated is
%   J = sum( (Y-R)^2 ) + RHO * sum( U^2 )
% where R is the reference, Y the plant output and U the control
% signal. (This due to change)
%
% If the global variable RUNSIM_BATCH_COUNT is set to a positive
% integer RUNSIM will be called with all initialitaion parameters
% in the interval 1:RUNSIM_BATCH_COUNT.
%
% [J,Y,U] = FMINCOST( SW, S1, S2, NR, NU, NY, RHO )
% This alternatively function call will also output the resulting
% output Y and control U.

global fmincost_COUNT RUNSIM_BATCH_COUNT

% optional arguments
if nargin < 8
  Nmax = 100;
end
if nargin < 9
  doplot = 0;
end

% function call count
if ( doplot ≥ 0 ) | ( fmincost_COUNT == [] )
  fmincost_COUNT = 0;
else
  fmincost_COUNT = fmincost_COUNT + 1;
end

% check for periodic plots
if (doplot < 0) & ( mod(fmincost_COUNT,10) == 0 ),
  doplot = -doplot;
end

% create neural network with lag network:
size_U = plant( 'inputcount' );
size_Y = plant( 'outputcount' );
global Y_POS
if Y_POS ~= []
  size_Y = size(Y_POS,2);
end
lagnn.R = zeros( 1, n_R );
lagnn.U = zeros( 1, size_U*n_U );
lagnn.Y = zeros( 1, size_Y*n_Y );

% Get neural network weights from scalar
[lagnn.W1, lagnn.W2] = scalar2weights( scalar, S1, S2, 1 );

% run simulation with lagnn
if (RUNSIM_BATCH_COUNT ~= []) & (RUNSIM_BATCH_COUNT > 0)
```

```
    for batch_count=1:RUNSIM_BATCH_COUNT
      [Y, U] = runsim( lagnn , Nmax, zeros(size_U), doplot , ...
                        batch_count );
    end
  else
    [Y, U] = runsim( lagnn , Nmax, zeros(size_U), doplot );
  end

% calculate costfunction
  cost = 1;
% Y = [ s d theta alpha_1 alpha_2 ]
  switch cost
   case {1,2,3}
    % cost = 1: 'standard' <-- use this
    % cost = 2: include traveled distance (use for start guess)
    % cost = 3: exclude penalty on angles outside bounds (pi/2)
    DeltaU=U(2:end)-U(1:end-1);
    remainingS = Y(end,1);
    if remainingS < 0,
      remainingS = 0;
    end
    J = ( 10*sum(Y(:,2).^2) + ...
          0.2*sum(Y(:,3).^2) + ...
          0.2*sum(Y(:,4).^2) + ...
          (cost~=3)*sum( ( Y(:,4) / (0.95*pi/2) ).^40 ) + ...
          (cost~=3)*sum( ( Y(:,5) / (0.95*pi/2) ).^40 ) ...
          ) / size(Y,1) + ...
        (cost==2) * remainingS + ...
        rho * sum( DeltaU.^2 ) / size(Y,1);
   case 4
    % for startguess (disregard distance to path)
    J = ( sum(Y(:,3).^2) + ...
          sum(Y(:,4).^2) + ...
          sum( ( Y(:,4) / (0.95*pi/2) ).^40 ) + ...
          sum( ( Y(:,5) / (0.95*pi/2) ).^40 ) ...
          ) / size(Y,1) + ...
        Y(end,1);
   otherwise
    J = 0;
    warning( 'No costfunction specified in fmincost.m' );
  end
```

A.2.5 runsim.m

```
  function [Y, U] = runsim( lagnn , Nmax, U_0, doplot , init_arg )
  % RUNSIM   Simulate neural network controller
  %
  % [Y, U] = RUNSIM( LAGNN )  Returns the resulting ouput into Y and
  % the resulting controlls into U. R is a (global) vector of
```

```
% reference signals and LAGNN is the neural controller with lag
% network. See LAGNNOUT for details.
%
% [Y, U] = RUNSIM( LAGNN, NMAX ) Additionally specifies the
% maximum number of iterations to simulate (default = 100).
%
% [Y, U] = RUNSIM( LAGNN, NAMX, UINIT) Does the same as above, but
% specifies UINIT as the inital plant and controller output.
%
% [Y, U] = RUNSIM( LAGNN, NAMX, UINIT, INITARG ) Additionally
% specifies an argument INITARG to be passed to the plant
% initialize function.
%
% The GLOBAL variable Y_POS should be used to limit certain plant
% output variables not to be used. See LAGNNOUT.

global Y_POS

if nargin < 2,
  Nmax = 100;
end

if nargin < 5
  plant('init'); % initialize plant model
else
  plant('init', init_arg); % initialize plant model
end

if nargin < 3
  U_0 = zeros( size(lagnn.W2,1), 1);
end

if nargin < 4
  doplot = 0;
end
if doplot > 1
  doderiv = 1; % calculate (and plot) derivates
else
  doderiv = 0;
end


% setup simulation
U(1,:) = U_0;
Y(1,:) = plant('actuate', U_0);
if Y_POS == [],
  Y_POS = 1:size(Y,2);
  clear_y_pos = 1;
else
```

```
  clear_y_pos = 0;
end

% run simulation
n = 2;
s = 0;
S = bezierlength;
progresscount=ceil(Nmax/20);
while ( (n < Nmax) & (s ≥ 0) & (s ≤ S) ),
  if mod(n,progresscount) == 0,
    fprintf('.');
  end
%   fprintf('%d',n);
  if doderiv == 1,
    [lagnn,U_n,YD_n] = lagnnout(lagnn, [], Y(n-1,:), U(n-1,:), ...
                                Y_POS);
   YD(n,:)=YD_n;
  else
    [lagnn,U_n] = lagnnout(lagnn, [], Y(n-1,:), U(n-1,:), Y_POS);
  end
  U(n,:)=U_n;
  Y(n,:)=plant('actuate', U(n));
  % get position on path (to check for end of simulation)
  s = Y(n,1);
  n = n + 1;
end
N = n - 1;
fprintf('\r                      \r');
if (clear_y_pos == 1)
  clear Y_POS;          % Clean up Y_POS
end


if ( nargin >2 ) & ( doplot > 0)
  if 0,
    nplots=2;
    if doderiv == 1,
      nplots=nplots+1;
    end

    % number of simulation step to plot
    N_max = 700;
    N_max = min(N_max, N); % limit to N
    n=1:N_max;
    yrow = 2; % row in y-matirx to plot
    yrow2 = 3; % row in y-matirx to plot

    subplot(nplots,1,1);
    plotyy(n,Y(1:N_max,yrow),n,Y(1:N_max,yrow2)/pi);
```

```
      legend('d',0);

      subplot(nplots,1,2);
      plot(n,U(1:N_max));
      legend('u',0);

      if doderiv == 1,
        subplot(nplots,1,3);
        plot(n,YD(1:N_max,:));
        legend('dy',0);
      end
    else
      plant('plot');
    end
    drawnow;
end
```

A.2.6 lagnnout.m

```
function [lagnn, U_new, YD] = lagnnout( lagnn, R, Y, U, YPOS )
% LAGNNOUT   Feed reference to lag-network and get NN output
%
% [LAGNN, U_NEW] = LAGNNOUT( LAGNN, R, Y, U )
%   lagnn is a cell array containing the lag network and the NN
%   weights. The updated lagnn is output to lagnn_new.
%   R is the reference; U is the current NN output and Y the
%   current plant output to be fed into the lag network.
%   U_new is the resulting NN output.
%
%          ###   #######    ########
%- --R----# #--#      #    #       #
%          #L#   #     #---#       #
%   /--Y--#A#--# NN  # | # Plant#------Y--
%    |     #G#   #     #-*-#       #   |
%    | /-U-# #--#      # | #       #   |
%    | |   ###   ####### | ########    |
%    | |                 |             |
%    |\-----------------/              |
%    \--------------------------------/
%
% lagnn.R is a vector containing the lag network for the
% references. size(lagnn.R) = n_R*size(R) where n_R is the number
% of references to keep in the lag network. Equivialently for
% lagnn.U and lagnn.Y containing the control and plant output lag
%   network.
% lagnn.W1 and lagnn.W2 contain the weights of the neural
% network. See nnout for details.
%
% [LAGNN, U_NEW, YD] = LAGNNOUT( LAGNN, R, Y, U )
% Also returns the partial derivatives of the neural network
% outputs with respect to the neural network inputs.
%
% [LAGNN, U_NEW] = LAGNNOUT( LAGNN, R, Y, U, YPOS )
% The optional YPOS parameter specifies the positions of Y to
% use. If the plant output Y is of size 1x4 then YPOS=[2 3] will
% ensure that only the 2nd and 2rd element in Y will be stored in
% the lag network and hence fed to the neural network.


debug = 0;
if not( debug )
  % shape the plant output signal
  if (nargin > 4),
    Y=Y(YPOS);
  end
  % update the lag network:
  lagnn.R=[R lagnn.R((size(R,2)+1):end)];
  if (nargin > 2),
    lagnn.U=[U lagnn.U((size(U,2)+1):end)];
  end
  if (nargin > 3),
    lagnn.Y=[Y lagnn.Y((size(Y,2)+1):end)];
  end

  % get the nn output
  U_new = nnout( [lagnn.R lagnn.Y lagnn.U]', lagnn.W1, lagnn.W2 );
  if nargout > 2,
    YD = nnderiv( [lagnn.R lagnn.Y lagnn.U]', lagnn.W1, lagnn.W2 );
  end
else
  % debug
  U_new = R;
  U_new = 0;
end
```

A.2.7 nnout.m

```
function Y = nnout( X, W1, W2 )
% NNOUT   Calculate neural network output
%
% Y = NNOUT( X, W1, W2 ) Returns in Y the outputs of a neural
% network with size( W1, 1 ) = size( W2, 2 )-1 hidden units with a
% tanh activation funtion. The network input is given by X and W1
% and W2 are the network weights and offsets.
%
% W1(:,1) are the offsets of the hidden unit inputs and likewise
% W2(:,1) are the offsets of the output units.
```

```
%
% The ouput is calculated corresponding to
%   Z = tanh( W1 * [1;X] );
%   Y = W2 * [1;Z];
% where Z are the hidden units.
%
% X  : inputs                     n_x * 1
% W1 : 1st layer weights          n_z * (n_x + 1)
% Z  : hidden layer units input   n_z * 1
% W2 : 2nd layer weights          n_y * (n_z + 1)
% Y  : output(s)                  n_y * 1

Y = W2 * [ 1;tanh( W1 * [1;X] )];
```

A.2.8 plant.m

```
function result=plant( action , input)
% PLANT  Simulate plant
%
% RESULT = PLANT( ACTION, INPUT) Simulates the plant. action
% specifies the action to be taken and can be one of the following:
% o 'init' Initialise plant. The must be called before using
%   'measure' or 'actuate'. No input nor return value.
% o 'savestate' Returns the current state in RESULT.
% o 'restorestate' Restores the state from the value passed in
%   INPUT, which in turn should be obtained by an (earlier) call to
%   'savestate'
% o 'measure' Returns in RESULT the current measurement. This does
%   not affect the plant's state.
% o 'acutate' Takes INPUT as the control signal(s) to the plant and
%   acutates the plant. Plant states will be updated accordingly
%   and the plant output will be returned in RESULT.
% o 'outputcount' / 'inputcount' return in RESULT the number of
%   output and input signals respectively.
% o 'plot' (if defined) will take plant specific data as INPUT and
%   plot it.
%
% When simulating the plant, first call 'init' and then use
% 'actuate'. 'savestate' and 'restorestate' (if implemented) allow
% simulation to go back in time, if neccessary.
%
% Several global variables are maintained, all with names starting
% with 'plant'. The should be considered private to the function
% and be left unaltered.
%
% This implementation simulates a car with one trailer. It uses an
% (s,d) coordinate system (see SD2XY).
% The kinematic equations are:
%        s' = v0 ( cos( theta ) ) / ( 1 - curv( s ) d )
%        d' = v0 sin( theta )
%    theta' = v0 [ ( tan( alpha_1 ) ) / l_1 -
%                  ( curv( s ) cos( theta ) )
%                     / ( 1 - curv( s ) d ) ]
%  alpha_1' = v0 1 / ( cos( alpha_1 ) ) *
%               [ tan( alpha_2 ) / l_2 - sin( alpha_1 ) / l_1 ]
%  alpha_2' = u
% where
%     theta = alpha_0 - theta_t
% theta     is the angle between the vehicle and the curve tangent
% theta_t   is the angle of the tangent relative to the cartesian
%           coordinate system
% alpha_0   is the angle of the vehicle relative to the the
%           cartesian coordinate system

global plantV2 plantS plantD plantTheta plantA plantL
global plantTs plantU
global plantOptions
global plantStoreU plantStoreY

switch lower( action )
 case 'init'
  % initialise bezier path
  bezierinit(1000,10);

  reversing = 1;
  % initialise car and vehicle variables
  if nargin > 1,
    scenario = input * 1;
  else
    scenario = 0;
  end
  % set up start position and orientation
  plantTheta = 0;
  plantD = 0;
  if reversing == 0,
    plantS = 0;
  else
    plantS = bezierlength; % reversing
  end
  plantA = [0 0];
  switch scenario
   case 0
    % ignore
   case 1
    plantTheta = plantTheta + 0.2;
   case 2
    plantTheta = plantTheta - 0.2;
   case 3
```

```
      plantA(1) = -0.2;
      plantD = -1;
    case 4
      plantA(1) = 0.2;
      plantD = 1;
    otherwise
      warning( 'Initialization␣argument␣too␣large.␣Ingoring.' );
  end
  % set up additional variables
  if reversing == 0,
    plantV2 = 1.4;  % velocity
  else
    plantV2 = -1.4;  % velocity (we're reversing)
  end
  plantL = [ 2; 1 ]; % length of vehicles

  % check for VARIATATE. This is used to change the real parameters
  % by a relative value in order to check parameter variations.
  global VARIATE
  if size(VARIATE,1) == 1,
    switch VARIATE.parameter,
      case 1,
        plantL(1) = plantL(1) * VARIATE.change;
      case 2,
        plantL(2) = plantL(2) * VARIATE.change;
      case 3,
        plantV2 = plantV2 * VARIATE.change;
    end
  end

  plantTs = 0.5;    % sampling period
  plantOptions = odeset('MaxOrder', 3, ...
                        'RelTol', 1e-4, ...
                        'AbsTol', [1e-6 1e-6 1e-6 1e-6 1e-6], ...
                        'Refine', 1);  % options for ODE solver
  % variables to store plant input / output (for post mortem
  % plotting )
  plantStoreY = [];
  plantStoreU = [];

 case 'savestate'
  result=0;
  warning( 'savestate␣is␣not␣implemented' );

 case 'restorestate'
  warning( 'restorestate␣is␣not␣implemented' );

 case 'measure'
  result = [plantS plantD plantTheta];
  case 'actuate'
   % input contains the control
   plantU = input;
   % solve differential equations
   y0 = [plantS; plantD; plantTheta; plantA'];
   if 1,
     [T,Y] = ode15s( @plantDiffM, [0 plantTs], y0, ...
                     plantOptions  );
   else
     [T,Y] = eulerode( @plantDiffM, [0 plantTs], y0 );
   end
   % update states
   plantS = Y(end,1);
   plantD = Y(end,2);
   plantTheta = Y(end,3);
   plantA = Y(end,4:5);
   %yd0 = plantDiffM( 0, y0 ); % DEBUG
   %yde = plantDiffM( 0, Y(end,:)' ); % DEBUG
   %debug = [y0 yd0  Y(end,:)' yde] % DEBUG

   % return result
   result = [plantS plantD plantTheta plantA ...
             beziercurvature(plantS)];
   % store control signal and outputs
   n = size(plantStoreU,1);
   plantStoreU(n+1,:) = input;
   plantStoreY(n+1,:) = result;

  case 'outputcount'
   result = 6;

  case 'inputcount'
   result = 1;

  case 'plot'
   % plot the car with trailers and the path

   if nargin<2,
     animate = 0;
   else
     animate = ( input ~= 0 );
   end

   % wheel image
   width     = 1;
   wheelY = [ 0    0   ]*width;
   wheelX = [ -0.2 0.2 ]*width;
   % car-image
```

```
carlength = plantL(2);
carY = ([0 1 1 0.45 0.5 0.55 0 0] -0.5)*width;
carX = ([0 0 1 1     1.1 1     1 0])*carlength;
%trailer-image
 trailerlength = plantL(1);
 trailY = ([0 1 1 0.5 0.5 0.5 0 0] -0.5)*width;
 trailX = ([0 0 1 1    1.1 1    1 0])*trailerlength;

% new plot
newplot;
subplot(1,1,1);
%subplot(4,1,1);

% plot path with fixed aspect ration ( to avoid distortion )
S = bezierlength;
N = 200;
s=0:(S/N):S;
[x,y]=bezierxy(s);
plot(x,y,'k-');
set(gca, 'DataAspectRatio', [1 1 1], 'PlotBoxAspectRatio', ...
          [1 1 1] );

% color cycle
colors = 5; max=0.8; min=0;
colorcycle=[ ...
     (max:(min-max)/(colors -1):min)' ...
     (max:(min-max)/(colors -1):min)' ...
     ones(colors ,1) ...
           ];
colors = colors+1;
colorcycle=[colorcycle; 0 0 0];

% plot car movement
s = plantStoreY(:,1)';
d = plantStoreY(:,2)';
th = plantStoreY(:,3)';
a1 = plantStoreY(:,4)';
a2 = plantStoreY(:,5)';
% convert to cartesian coordinate system
[x,y] = sd2xy(s,d);     % coordinate
[xd,yd]=bezierxyd(s);   % get tangent
oldwarn=warning('off');
tht = (atan(yd./xd)).*(sign(xd)>=0) + ...
      (pi+atan(yd./xd)).*(sign(xd)<0); % tangent angle
warning(oldwarn);
a0 = th + tht;              % get absolute car angel
x1 = x + plantL(1)*cos(a0);
y1 = y + plantL(1)*sin(a0);
a01 = a0 + a1;
```

```
x2 = x1 + plantL(2)*cos(a01);
y2 = y1 + plantL(2)*sin(a01);
a02 = a01 + a2;

% plot paths
hold on
if 0,
  plot(x,y,'r');
  plot(x1,y1,'r--');
else
  plot(x,y,'k:');
  plot(x1,y1,'k');
end
hold off

% reduce dataset and plot vehicle
if ~animate,
  i = 1:5:size(s,2);
else
  i = 1:size(s,2);
end
x   = x(i);
y   = y(i);
a0  = a0(i);
x1  = x1(i);
y1  = y1(i);
a01  = a01(i);
x2  = x2(i);
y2  = y2(i);
a02  = a02(i);


%  pause

if animate,
  % animated plot
  trailerhandle = line( 0, 0, 'Color', 'k' );
  carhandle = line( 0, 0, 'Color', 'k' );
  wheelhandle = line( 0, 0, 'Color', 'k' );
  texthandle = text( 10, -10, '' );
  for n=1:size(x,2),
    set(texthandle, 'String', int2str(n));
    t = clock;
    rx = trailX*cos(a0(n)) - trailY*sin(a0(n)) + x(n);
    ry = trailX*sin(a0(n)) + trailY*cos(a0(n)) + y(n);
    set( trailerhandle, 'XData', rx );
    set( trailerhandle, 'YData', ry );
    rx = carX*cos(a01(n)) - carY*sin(a01(n)) + x1(n);
    ry = carX*sin(a01(n)) + carY*cos(a01(n)) + y1(n);
```

```
 %   rx = carX + x1(n);
 %   ry = carY + y1(n);
      set( wheelhandle, 'XData', rx );
      set( wheelhandle, 'YData', ry );
      rx = wheelX*cos(a02(n)) - wheelY*sin(a02(n)) + x2(n);
      ry = wheelX*sin(a02(n)) + wheelY*cos(a02(n)) + y2(n);
      set( carhandle, 'XData', rx );
      set( carhandle, 'YData', ry );
      drawnow;
      while etime( clock, t ) < 0.5
        ;
      end
    end
  end
  for colorcount=1:colors
    for n=colorcount:colors:size(x,2)
      if 0,
        plot(x(n),y(n),'bo');
        plot(x1(n),y1(n),'ro');
        plot(x2(n),y2(n),'mo');
      else
        rx = trailX*cos(a0(n)) - trailY*sin(a0(n)) + x(n);
        ry = trailX*sin(a0(n)) + trailY*cos(a0(n)) + y(n);
        line(rx, ry, 'Color', colorcycle(mod(n,colors)+1,:));
        rx = carX*cos(a01(n)) - carY*sin(a01(n)) + x1(n);
        ry = carX*sin(a01(n)) + carY*cos(a01(n)) + y1(n);
        line(rx, ry, 'Color', colorcycle(mod(n,colors)+1,:));
        rx = wheelX*cos(a02(n)) - wheelY*sin(a02(n)) + x2(n);
        ry = wheelX*sin(a02(n)) + wheelY*cos(a02(n)) + y2(n);
        line(rx, ry, 'Color', colorcycle(mod(n,colors)+1,:));
      end
    end
  end

  if 0,
    iter=1:size(plantStoreY,1);
    s = plantStoreY(:,1)';
    d = plantStoreY(:,2)';
    th = plantStoreY(:,3)';
    a1 = plantStoreY(:,4)';
    a2 = plantStoreY(:,5)';
    u = plantStoreU';

    figure
    subplot(3,1,1)
    plot(iter,d)
    legend('d',0)
    subplot(3,1,2)
    plot(iter,th,iter,a1,iter,a2)
```

```
      legend('\theta','\alpha_1','\alpha_2',0)
      subplot(3,1,3)
      plot(iter,u)
      legend('u',0)

      drawnow
      fprintf('press enter');
      pause
      fprintf('\r                \r');
    end

   otherwise
    error('Unknown plant action')

  end

  % sub-functions

  function dy = plantDiffM( t, y );
  global plantV2 plantU plantL
  curv = beziercurvature1( y(1,:) );
  c3omcd = cos( y(3,:) )./1 - curv .* y(2,:);
  c4 = cos( y(4,:) );
  c5 = cos( y(5,:) );
  v0 = plantV2 * c4 .* c5;
  dy = [ ...
         v0*c3omcd; ...
         v0*sin( y(3,:) ); ...
         plantV2 * c5 .* ...
      ( sin( y(4,:) ) ./ plantL(1) - curv .* c4 .* c3omcd ); ...
         plantV2 * ( sin( y(5,:) ) ./ plantL(2) - ...
                     sin( y(4,:) ) .* c5 ./ plantL(1) ); ...
         plantU
      ];

  function [T,Y] = eulerode( odefun, tspan, y0 ),
  % we'll assume, that tspan = [0 tend]:
  tend = tspan(end);
  steps=400;
  tstep = tend/steps;
  y = y0;
  for n = 1:steps;
    t = n*tstep;
    y = y+tstep*feval(odefun, t, y);
    T(n) = t;
    Y(n,:) = y';
  end
  % we really should add an entry for t=0...
```

A.2.9 sd2xy.m

```
function [x, y]=sd2xy( s, d )
% SD2XY  Convert path-relative coordinates into cartesian coord.
%
%  [X,Y] = SD2XY( S, D ) Converts the path-relative coordinates
%  (S,D) into the cartesian coordinates (X,Y). S is the distance
%  along the path and D the perpendicular distance to the path
%  (positive D correspond to right side of path).
%  The path has to be initialised be BEZIERINIT.

debug = 0;

% find coordinate of s
[x,y] = bezierxy(s);

% get tangent direction
[xd,yd] = bezierxyd(s);
% norm vector sqrt(xd.^2+yd.^2) = 1
l=sqrt(xd.^2+yd.^2);
xd = xd ./ l;
yd = yd ./ l;

x = x - d.*yd;
y = y + d.*xd;
```

A.2.10 bezierinit.m

```
function bezierinit( N, option )
% BEZIERINIT Initialise bezier variables
%
% BEZIERINIT will initialise the global variables needed for the
% other bezier functions to work. Data is calculated and stored to
% the file 'bezier.mat'. If it exists upon function call, the file
% will be read instead of re-calculation.
%
% BEZIERINIT( N ) The optional parameter N specifies the average
% points per bezier section to calculate and defaults to
% 1000. Higher N yield higher precision.
%
% NOTE: Current implementation will not include the last point of
% the bezier curve.

if nargin < 2,
  curve = 7;
else
  curve = option;
end
```

```
if nargin < 1,
  N=1000;
end

clear global BEZIER_X
clear global BEZIER_Y
global BEZIER_X BEZIER_Y
global BEZIER_S BEZIER_T BEZIER_DSDT

% bezier courve parameters
%
% Each row contains [position derivative] for the given curve
% point. X(n,:) and Y(n,:) specify the n'th curve point and
% derivative at that point. Unit is meters. Currently # 7 is used
% for training and # 9 for test.
switch curve
 case 1
  X = [ 0    1.5 ; 2   0    ; 2   3];
  Y = [ 0.5 1.5 ; 1  -1.5 ; 0   0];
 case 2  %45-degree line
  X = [ 0 1 ; 2 1 ];
  Y = [ 0 1 ; 2 1 ];
 case 3 % circle
  a = 1.675;
  X = [ 0 a ; 1 0 ; 0 -a ; -1 0   ; 0 a];
  Y = [ 0 0 ; 1 a ; 2  0 ;  1 -a ; 0 0];
 case 4 % S
  X = [ -1.5 1 ; -1 1 ; 0    0;  0    0 ;   1   1 ; 1.5 1 ]*15;
  Y = [      0 1 ; .5 1 ; .5 -2;  -.5  -2 ;  -.5 1 ; 0    1 ]*15;
 case 5 % 'soft' turn
  X = [ 0 1    ; 2  1    ];
  Y = [ 0 0.1 ; 0  -0.1 ];
 case 6 % 'hat'
  a = 16.75;
  X = [ 0 a ; 10 0 ; 20 a ; 30 a ; 40   0 ; 50 a ];
  Y = [ 0 0 ; 10 a ; 20 0 ; 20 0 ; 10  -a ; 0   0 ];
 case 7 % 4 times S (used for training)
  a = 16.75; b=a/2; c=a*2; d=a/5;
  X = [ -40 d ; -22 d ; -20   0 ; -20   0 ; -18 d ];
  Y = [    20 0 ;   20 0 ;  18 -d ;    2 -d ;     0 0 ];
  X = [ X ; 0 a ; 10 0 ; 10 0 ; 20 a ; 25 1 ; 30 b ; 35   0 ];
  Y = [ Y ; 0 0 ; 10 a ; 20 a ; 30 0 ; 30 0 ; 30 0 ; 25  -b ];
  X = [ X ; 35   0 ; 40 b ; 50 c ; 70       0 ; 90 c ];
  Y = [ Y ; 15 -b ; 10 0 ; 10 0 ; 30 c*0.7 ; 50 0 ];
 case 8
  X = [ 0 1 ; 19 2 ; 21   2 ; 38   2 ; 42 2 ; 60 2 ];
  Y = [ 0 1 ; 19 2 ; 19 -2 ;   2 -2 ;   2 2 ; 20 2 ];
 case 9
  a = 28; b = a/2; c = b/2;
```

```
    X = [    0 0 ;    1 0 ; 20 a ];
    Y = [ -20 c ; -10 a ; 20 a/10 ];
    X = [ X ; 35 c ; 40  0 ; 40  0 ; 35 -c ];
    Y = [ Y ; 20 0 ; 15 -c ;  5 -c ; -1  0 ];
    X = [ X ; 25 -b ;  15  0 ;  15  b/10 ];
    Y = [ Y ;   3  0 ; -11 -b ; -20 -b ];
  case 10
    a = 20; b = 40; c = 5;
    X = [ 0 a ; 20 0 ; 23 c ];
    Y = [ 0 0 ; 20 b ; 50 b ];
  case 17 % 4 times S (used for training)
    a = 16.75; b=a/2; c=a*2;
    X = [ 0 a ; 10 0 ; 10 0 ; 20 a ; 25 1 ; 30 b ; 35  0 ];
    Y = [ 0 0 ; 10 a ; 20 a ; 30 0 ; 30 0 ; 30 0 ; 25 -b ];
    X = [ X ; 35  0 ; 40 b ; 50 c ; 70     0 ; 90 c ];
    Y = [ Y ; 15 -b ; 10 0 ; 10 0 ; 30 c*0.7 ; 50 0 ];
  otherwise % x-axis
    X = [ 0 10 ; 20 10 ];
    Y = [ 0  0 ;  0  0 ];
end


% check if data saved to file is the same
try
  load bezier;
end
% the 1st IF line is a path for the second when dimensions don't
% agree. It will trigger the second IF.
if sum( size(BEZIER_X) ~= size(X) ) | ...
      sum( size(BEZIER_Y) ~= size(Y) ),
  BEZIER_X = rand(size(X));
  BEZIER_Y = rand(size(Y));
end
if sum(sum( BEZIER_X ~= X )) | sum(sum( BEZIER_Y ~= Y )) | ...
      (size(BEZIER_S,2) ~= N)
  % Data in file differs
  % Calculate new curve data

  % copy bezier curve parameters and clear old arrays
  BEZIER_X = X;
  BEZIER_Y = Y;
  clear global BEZIER_S
  clear global BEZIER_T
  clear global BEZIER_DSDT
  clear global BEZIER_A
  clear global BEZIER_B
  global BEZIER_A BEZIER_B
  global BEZIER_a BEZIER_b
  global BEZIER_S BEZIER_T BEZIER_DSDT
```

```
sections = size(BEZIER_X,1)-1;
options = optimset;

% Calculate total path length
S = 0;
for section=1:sections,
  % get bezier curve parameters for this section
  x0  = BEZIER_X( section,   1 );
  xd0 = BEZIER_X( section,   2 );
  x1  = BEZIER_X( section+1, 1 );
  xd1 = BEZIER_X( section+1, 2 );
  y0  = BEZIER_Y( section,   1 );
  yd0 = BEZIER_Y( section,   2 );
  y1  = BEZIER_Y( section+1, 1 );
  yd1 = BEZIER_Y( section+1, 2 );
  BEZIER_a = [x0 xd0 (3*(x1-x0)-(2*xd0+xd1)) ...
              (2*(x0-x1)+xd0+xd1) ];
  BEZIER_b = [y0 yd0 (3*(y1-y0)-(2*yd0+yd1)) ...
              (2*(y0-y1)+yd0+yd1) ];
  BEZIER_A( section, : ) = BEZIER_a;
  BEZIER_B( section, : ) = BEZIER_b;

  sectionLength(section) = pathLength( 1 );
  S = S + sectionLength(section);
end

% start with section 1
section = 1;

% get bezier curve parameters for this section
x0  = BEZIER_X( section,   1 );
xd0 = BEZIER_X( section,   2 );
x1  = BEZIER_X( section+1, 1 );
xd1 = BEZIER_X( section+1, 2 );
y0  = BEZIER_Y( section,   1 );
yd0 = BEZIER_Y( section,   2 );
y1  = BEZIER_Y( section+1, 1 );
yd1 = BEZIER_Y( section+1, 2 );
BEZIER_a = [x0 xd0 (3*(x1-x0)-(2*xd0+xd1)) (2*(x0-x1)+xd0+xd1) ];
BEZIER_b = [y0 yd0 (3*(y1-y0)-(2*yd0+yd1)) (2*(y0-y1)+yd0+yd1) ];
BEZIER_A( section, : ) = BEZIER_a;
BEZIER_B( section, : ) = BEZIER_b;

% Determine how often to update progress
nCount=floor(N/100);
if nCount == 0, nCount = 1; end

for n=1:N,
    if mod(n,nCount) == 0,
      fprintf(['initialising␣courve␣data␣for␣segment␣%d␣with␣' ...
               '%d␣steps:␣%3d%%␣done\r'], ...
              section, N, round(100*(n-1)/(N-1)));
    end
    % position along segment
    s=S*(n-1)/(N-1);
    BEZIER_S(n) = s;
    % check if we entered the next section
    if (s > sum(sectionLength(1:section))) & (section<sections),
      section = section + 1;
      % get bezier curve parameters for this section
      x0  = BEZIER_X( section ,   1 );
      xd0 = BEZIER_X( section ,   2 );
      x1  = BEZIER_X( section+1 , 1 );
      xd1 = BEZIER_X( section+1 , 2 );
      y0  = BEZIER_Y( section ,   1 );
      yd0 = BEZIER_Y( section ,   2 );
      y1  = BEZIER_Y( section+1 , 1 );
      yd1 = BEZIER_Y( section+1 , 2 );
      BEZIER_a = [x0 xd0 (3*(x1-x0)-(2*xd0+xd1)) ...
                  (2*(x0-x1)+xd0+xd1) ];
      BEZIER_b = [y0 yd0 (3*(y1-y0)-(2*yd0+yd1)) ...
                  (2*(y0-y1)+yd0+yd1) ];
      BEZIER_A( section , : ) = BEZIER_a;
      BEZIER_B( section , : ) = BEZIER_b;
    end
    % find corresponding t
    oldwarn = warning('off');
    t0 = (s-sum(sectionLength(1:(section -1)))) / ...
         sectionLength(section);
    t = fzero(@distance,t0,options, ...
              s-sum(sectionLength(1:(section -1))));
    warning(oldwarn);
    BEZIER_T(n) = t + (section - 1);
    % estimate ds/dt
    if n == 1,
    elseif n == 2,
      BEZIER_DSDT(n-1) = ...
          (BEZIER_S(n-1)-BEZIER_S(n))/(BEZIER_T(n-1)-BEZIER_T(n));
    else
      BEZIER_DSDT(n-1) = ...
          0.5*(BEZIER_S(n-1)-BEZIER_S(n)) / ...
          (BEZIER_T(n-1)-BEZIER_T(n)) + ...
          0.5*(BEZIER_S(n-1)-BEZIER_S(n-2)) / ...
          (BEZIER_T(n-1)-BEZIER_T(n-2));
      if n == N,
        BEZIER_DSDT(n) = (BEZIER_S(n)-BEZIER_S(n-1)) / ...
            (BEZIER_T(n)-BEZIER_T(n-1));
        end
      end
    end

   % save variables to file
    fprintf('saving␣to␣file\r');
    save bezier BEZIER_X BEZIER_Y BEZIER_A BEZIER_B ...
        BEZIER_S BEZIER_T BEZIER_DSDT

    fprintf(['␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣' ...
             '␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\r']);

   % do post clean-up
    clear global BEZIER_a
    clear global BEZIER_b
  end


  function d=distance(t,s)
  d = pathLength(t)-s;

  function s=pathLength(t)
  s = quad(@speed,0,t);

  function sd=speed(t)
  % we're using some global variables here to speed things up
  global BEZIER_a BEZIER_b

  sd = sqrt( ...
      (BEZIER_a(2) + BEZIER_a(3) * 2*t + BEZIER_a(4) * 3*t.^2).^2 + ...
      (BEZIER_b(2) + BEZIER_b(3) * 2*t + BEZIER_b(4) * 3*t.^2).^2 ...
      );
```

A.2.11 bezierxy.m

```
  function [x,y] = bezierxy( s )
  % BEZIERXY  Calculate x- and y-coordinate of curve point
  %
  % [X,Y] = BEZIERXY( S ) Returns in X the x-coordinate of the bezier
  % curve initialised by BEZIER_INIT. S is the distance along the
  % bezier curve from the beginning of the curve to the point.

  global BEZIER_S BEZIER_T BEZIER_DSDT
  global BEZIER_A BEZIER_B

  % find the position closest to s in the arrays:
  % We assume BEZIER_S(1)=0;
  N = size(BEZIER_S,2);
  S = BEZIER_S(N);
```

```
n = round(s/S*(N-1))+1;
limitAt = n<1;
n = n.*(1-limitAt)+limitAt;
limitAt = n>N;
n = n.*(1-limitAt)+N.*limitAt;

% get estimate for t
dsdt = BEZIER_DSDT(n);
t = BEZIER_T(n) + (s - BEZIER_S(n))./dsdt;

% get section and limit
section = floor(t)+1;
limit = 1;
limitAt = section<limit;
section = section.*(1-limitAt)+limit.*limitAt;
limit = size(BEZIER_A,1);
limitAt = section>limit;
section = section.*(1-limitAt)+limit.*limitAt;

t = t-(section -1);
T = [ ones(size(t)); t; t.^2; t.^3 ];

x = sum(BEZIER_A(section,:) .* T',2)';
y = sum(BEZIER_B(section,:) .* T',2)';
```

A.2.12 bezierxyd.m

```
function [xd,yd] = bezierxyd( s )
% BEZIERXYD  Calculate derivative of curve point
%
% [XD,YD] = BEZIERXY( S ) Returns in XD and YD the tangent of the
% bezier curve initialised by BEZIER_INIT. S is the distance along
% the bezier curve from the beginning of the curve to the point. XD
% and YD are the derivates with respect to the parameter t, such
% that XD = dx(t)/dt and YD = dy(t)/dt, where t is defined by
%   s = \int^t_0 \sqrt{x'^2(\tau) + y'^2(\tau)} d\tau

global BEZIER_S BEZIER_T BEZIER_DSDT
global BEZIER_A BEZIER_B

% find the position closest to s in the arrays:
% We assume BEZIER_S(1)=0;
N = size(BEZIER_S,2);
S = BEZIER_S(N);
n = round(s/S*(N-1))+1;
limitAt = n<1;
n = n.*(1-limitAt)+limitAt;
limitAt = n>N;
n = n.*(1-limitAt)+N.*limitAt;
```

```
% get estimate for t
dsdt = BEZIER_DSDT(n);
t = BEZIER_T(n) + (s - BEZIER_S(n))./dsdt;

% get section and limit
section = floor(t)+1;
limit = 1;
limitAt = section<limit;
section = section.*(1-limitAt)+limit.*limitAt;
limit = size(BEZIER_A,1);
limitAt = section>limit;
section = section.*(1-limitAt)+limit.*limitAt;

t = t-(section -1);
T = [ zeros(size(t)); ones(size(t)); 2*t; 3*t.^2 ];

xd = sum(BEZIER_A(section,:) .* T',2)';
yd = sum(BEZIER_B(section,:) .* T',2)';
```

A.2.13 bezierxydd.m

```
function [xdd,ydd] = bezierxydd( s )
% BEZIERXYDD  Calculate second derivative of curve point
%
% [XDD,YDD] = BEZIERXY( S ) Returns in XDD and YDD the second
% derivative of the point on the bezier curve initialised by
% BEZIER_INIT. S specifies the point and is the distance along the
% bezier curve from the beginning of the curve to the point. XDD
% and YDD are the derivates with respect to the parameter t, such
% that XD = dx(t)/dt and YD = dy(t)/dt, where t is defined by
%   s = \int^t_0 \sqrt{x'^2(\tau) + y'^2(\tau)} d\tau

global BEZIER_S BEZIER_T BEZIER_DSDT
global BEZIER_A BEZIER_B

% find the position closest to s in the arrays:
% We assume BEZIER_S(1)=0;
N = size(BEZIER_S,2);
S = BEZIER_S(N);
n = round(s/S*(N-1))+1;
limitAt = n<1;
n = n.*(1-limitAt)+limitAt;
limitAt = n>N;
n = n.*(1-limitAt)+N.*limitAt;

% get estimate for t
dsdt = BEZIER_DSDT(n);
```

```
t = BEZIER_T(n) + (s - BEZIER_S(n))./dsdt;

% get section and limit
section = floor(t)+1;
limit = 1;
limitAt = section<limit;
section = section.*(1-limitAt)+limit.*limitAt;
limit = size(BEZIER_A,1);
limitAt = section>limit;
section = section.*(1-limitAt)+limit.*limitAt;

t = t-(section-1);
T = [ zeros(size(t)); zeros(size(t)); 2*ones(size(t)); 6*t ];

xdd = sum(BEZIER_A(section,:) .* T',2)';
ydd = sum(BEZIER_B(section,:) .* T',2)';
```

A.2.14 bezierlength.m

```
function s = bezierlength
% BEZIERLENGTH  Return total length of bezier path
%
% S = BEZIERLENGTH returns in S the total length of the bezier
% path initialised by BEZIERINIT.

global BEZIER_S

s = BEZIER_S( end );
```

A.2.15 beziercurvature.m

```
function [K] = beziercurvature( s )
% BEZIERCURVATURE Calculate x- and y-coordinate of curve point
%
% [K] = BEZIERCURVATURE( S ) Returns in K the curvature of the
% bezier path at distance s from the starting point. The bezier
% path must be initialised by BEZIERINIT.

global BEZIER_S BEZIER_T BEZIER_DSDT
global BEZIER_A BEZIER_B

%[xd, yd] = bezierxyd( s );
%[xdd, ydd] = bezierxydd( s );

% find the position closest to s in the arrays:
% We assume BEZIER_S(1)=0;
N = size(BEZIER_S,2);
S = BEZIER_S(N);
n = round(s/S*(N-1))+1;
```

```
limitAt = n<1;
n = n.*(1-limitAt)+limitAt;
limitAt = n>N;
n = n.*(1-limitAt)+N.*limitAt;


% get estimate for t
dsdt = BEZIER_DSDT(n);
t = BEZIER_T(n) + (s - BEZIER_S(n))./dsdt;

% get section and limit
section = floor(t)+1;
limit = 1;
limitAt = section<limit;
section = section.*(1-limitAt)+limit.*limitAt;
limit = size(BEZIER_A,1);
limitAt = section>limit;
section = section.*(1-limitAt)+limit.*limitAt;

t = t-(section -1);

st = size(t);
T = [ zeros(st); ones(st); 2*t; 3*t.^2 ]';
xd = sum(BEZIER_A(section,:) .* T,2);
yd = sum(BEZIER_B(section,:) .* T,2);

T = [ zeros(st); zeros(st); 2*ones(st); 6*t ]';
xdd = sum(BEZIER_A(section,:) .* T,2);
ydd = sum(BEZIER_B(section,:) .* T,2);

% calculate and return curvature

K = (xd.*ydd-yd.*xdd)./((xd.^2+yd.^2).^1.5);
```

Bibliography

[1] D. Clarke, C. Mohtadi, and P. Tuffs, "Generalized predictive control - part i. the basic algorithm," *Automatica*, vol. 23, no. 2, pp. 137–148, 1987.

[2] D. Clarke, C. Mohtadi, and P. Tuffs, "Generalized predictive control - part ii. extensions and interpretations," *Automatica*, vol. 23, no. 2, pp. 149–160, 1987.

[3] J. Sjöberg, "Some aspects of neural nets and related model structures for nonlinear system identification," Tech. Rep. CTH-TE-70, Chalmers University of Technology, 412 96 Gothenburg, Sweden, March 1998.

[4] M. Nørgaard, O. Ravn, N. K. Poulsen, and L. K. Hansen, *Neural Networks for Modelling and Control of Dynamic Systems*. Springer-Verlag, 2000.

[5] C. M. Bishop, *Neural Networks for Pattern Recognition*. Oxford University Press, 1995.

[6] N. K. Poulsen, *Stochastic Adaptive Control - Exercise 1*. IMM, DTU, march 2002.

[7] L. Skovgaard, "Regulering af ustabilt ulineært objekt," Master's thesis, Danmarks Tekniske Universitet, August 2000.

[8] C. Samson, "Control of chained systems application to path following and time-varying point-stabilization of mobile robots," *IEEE Transactions on Automatic Control*, vol. 40, pp. 64–77, January 1995.

Index