NNCTRL - a CANCSD toolkit for MATLAB(R)

Nørgård, Peter Magnus; Ravn, Ole; Poulsen, Niels Kjølstad; Hansen, Lars Kai

Published in:
Proceedings of the 1996 IEEE Symposium on Computer-Aided Control System Design

Link to article, DOI:
10.1109/CACSD.1996.555320

Publication date:
1996

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

Proceedings of the 1996 IEEE International Symposium
on Computer-Aided Control System Design
Dearborn, MI • September 15-18, 1996

NNCTRL - A CANCSD Toolkit for MATLAB®

M. Nørgaard*, O. Ravn*, N.K. Poulsen**, & L.K. Hansen**

*Department of Automation, building 326. pmn, or@iau.dtu.dk
**Department of Mathematical Modelling, building 321. nkp, lkh@imm.dtu.dk
Technical University of Denmark (DTU), 2800 Lyngby, Denmark

Abstract

A set of tools for Computer-Aided Neuro-Control System Design (CANCSD) has been developed for the MATLAB environment. The tools can be used for construction and simulation of a variety of neural network based control systems. The design methods featured in the toolkit are: direct inverse control, internal model control, feedforward, feedback linearization, optimal control, instantaneous linearization, and nonlinear predictive control. Furthermore, the toolkit has been given a flexible design which allows for incorporation of the user's personal control algorithms.

1. Introduction

In this paper we discuss an engineering tool for design of control systems for processes which are hard to model in a deductive fashion. Our approach to the problem has been the typical system identification approach: 1) Conduct an experiment with the process to acquire a set of data and use this to infer a model of the process. 2) Design a control system for the identified model. Whenever possible, it is recommended to rely on the linear system identification techniques [7] followed by a conventional controller design [3]. However, when this strategy fails to work, neural networks are often recommended instead ([16], [5]). This is due to their excellent ability to model nonlinear systems, which has been reported on several occasions [15]. While there seemed to be a lack of generic tools for neural network based control system design we decided to develop some ourselves. This paper presents the result of our efforts - *The NNCTRL toolkit.*

Having decided to attempt a neural network strategy for solving a particular control problem, different approaches are possible:

- Model based *or* not model based.
- A network used directly as the controller *or* an indirect design based on a neural network model of the process.

While there may be opposing opinions on whether or not it is a good idea to let a neural network implement the actual controller, most strategies are model based and thus require a neural network model of the process. A typical working procedure for design of model based controllers with neural networks can be divided into the following three major tasks (which are not necessarily performed independently):

- System Identification. That is, to infer a process model from a set of data collected in an experiment.
- Based on the identified model, construct a controller (which might be a neural network as well) and simulate the closed-loop system. This is done in order select a suitable control design and to tune the design parameters.
- Implementation in a real-time system and application to the real process.

While the NNSYSID toolbox described in [10] was designed explicitly for solving the system identification task, the NNCTRL toolkit presented in this paper has been developed to assist the control engineer in solving the second task. The toolkit is developed in MATLAB due the excellent data visualization features and its support for simulation of dynamic systems. It has also been a motivation that MATLAB is extremely popular in the control engineering community. Apart from the NNSYSID toolbox the toolkit requires the Signal Processing toolbox provided by the MathWorks, Inc. Although not a vital requirement it is also an advantage if SIMULINK® is available. If it is not present MATLAB's build-in ODE solver is used instead.

The toolkit has been given a structure that facilitates incorporation of new control concepts. This is an attractive feature if the ones provided in advance are not sufficent for the problem under consideration, or if the user simply would like to test new ideas. Furthermore, since all the code is written as 'm-files', it is very easy to understand and modify the existing code if desired.

The paper begins by describing the fundamental program structure shared by the different neural network based control systems. The functions in the toolkit are then presented by category according to type of control system to which they belong. Finally some comments on the real-time perspectives are given.

2. The Basic Concept

All the different control systems have been implemented within the same framework. This framework contains different standard components, such as reading a design parameter file, variable initialization, reference signal generation, process simulation, a constant gain PID controller, data storage, plots, and more. The control systems can be divided into two basic categories:

- The controller is implemented by a neural network.
- An indirect design based on a neural network model of the process. The controller is not itself a neural netvork.

The fundamental program structure is slightly different for each category. The program structures are shown in fig. 1.

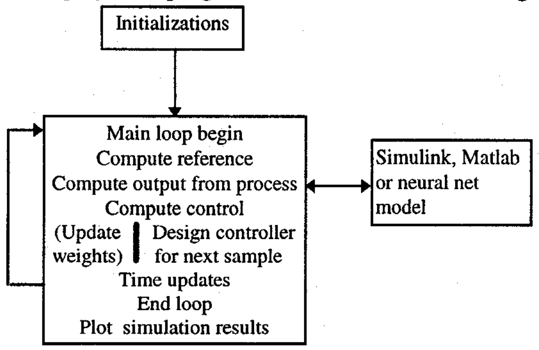

Figure 2. Program structures. Indirect design: a controller is designed for a pretrained network at each sample. Direct design: The controller is often trained online and for this reason the "update weights" stage is included. When the network is properly trained, this stage is omitted in the final use.

Each of the boxes in fig. 1 symbolizes a MATLAB script file or function. The three basic components are:

- A function describing the process to be controlled. The process can be specified as either a SIMULINK model, a MATLAB function containing the differential equations, or a neural network model of the process. The SIMULINK and MATLAB options are of course only relevant when a physical model exists.

  A MATLAB script file containing initializations specified by the user. Some initializations typically required are: choice of reference signal, sampling frequency, name of Simulink/MATLAB function implementing the process, PID or neural network based controller, design parameters for the controller. This file has a prespecified name and format associated with the type of control system. The name is always concluded by the letters *init.m* (for example it is called *invinit.m* in direct inverse control and *npcinit.m* in nonlinear predictive control). When working with NNCTRL a "template" initialization file is copied to the working directory and modified to comply with the application under consideration.
- The main program which automatically reads the initialization file and simulates the process. This program usually contains the letters *con.m* in its name to emphasize that it is the control system simulation program (for example *invcon.m* for direct inverse control).

3. Implemented Control Systems

In this chapter the different types of controllers implemented in the NNCTRL toolkit are briefly explained. This includes: control with inverse models, feedforward, input-output linearization, optimal control, controllers based on instantaneous linearization, and nonlinear predictive control.

Control with Inverse Models

The most fundamental neural network based control system designs are propably those using the "inverse" of the process as the controller. The idea is, that if the process can be described by

$$y(t+1) = g(y(t),\ldots,y(t-n+1),u(t),\ldots,u(t-m))$$

a network is trained as the inverse of the process:

$$\hat{u}(t) = \hat{g}^{-1}(y(t+1),y(t),\ldots,y(t-n+1),u(t-1),u(t-m))$$

The inverse model is subsequently applied as controller for the proces by inserting the desired output, the *reference* $r(t+1)$, instead of the output $y(t+1)$. There are several references on this principle. See for example [12], [4], and [5].

In relation to the training of inverse models and simulation of control systems incorporating these, the toolkit provides the following files:

general	General training of inverse models.
special1	Specialized back-prop training of inverse models.
invinit1	File with design parameters for *special1*.
special2	Specialized training with simplified recursive Gauss-Newton method.
invinit2	File with design parameters for *special2*.
special3	Specialized training, recursive Gauss-Newt. met.
invsim	Function for evaluating inverse models.
invcon	Program for simulating direct inverse control.
invinit	File with design parameters for *invcon*.
imccon	Program for simulating internal model control.
imcinit	File with design parameters for *imccon*.
ffcon	Program for simulating PID+feedforward control.
ffinit	File with design parameters for *ffcon*.

Two methods have been implemented to support the establishment of inverse models: general training and specialized training [12]. In *general training* a network is trained off-line to minimize the criterion (θ specifies the network weights):

$$J_1(\theta) = \sum_{t=1}^{N} \left(u(t) - \hat{u}(t|\theta)\right)^2$$

An experiment is performed and a set of corresponding inputs and outputs are stored. Subsequently the function *general,* which applies a version of the Levenberg-Marquardt method [1], is invoked.

Specialized training is an on-line algorithm related to model-reference adaptive control. The idea is minimize the criterion:

$$J_2(\theta) = \sum_t \left(r(t) - y(t)\right)^2$$

Before the training of the inverse model is initiated a model of the process is required. This is created with the NNSYSID toolbox [10] from a data set collected in an initial experiment. Unlike general training, the control design is thus "model-based" when specialized training is applied to build the inverse model. Details on the principle can be found in [4]. Three different versions of the scheme have been implemented: One using a recursive back-propagation algorithm for minimizing the criterion (*special1*), a more rapid that is using a recursive Gauss-Newton algorithm (occationally referred to as a recursive prediction error method, [7]) (*special3*), and one that uses a simplified recursive Gauss-Newton algorithm (*special2*). The typical working procedure is to initialize the inverse model with general training and the proceed with specialized training to fine-tune the network.
Once an inverse model has been trained, there are different ways in which it can be used for control purposes. The toolkit provides three different concepts:

Direct inverse control:
As the name indicates, the inverse model is used as the controller directly.

$$u(t) = \hat{g}^{-1}\left(r(t+1), y(t), \ldots, y(t-n+1), u(t-1), u(t-m)\right)$$

invcon is used for simulating the closed-loop system, while design parameters are specified in the file *invinit.*

Internal model control:
This design is discussed in [4]. The control signal is synthesized from a "forward" model of the process *as well* an inverse model. An attractive property of this design is that it produces an off-set free steady-state response despite the process is affected by a constant disturbance. It is implemented in the file *imccon.*

Feedforward
Using inverse models for feedback control leads to a dead-beat type control, which is unsuitable in most cases. If a PID-controller already has been tuned for stabilizing the process (which is common in many industrial applications), an inverse model used as feedforward controller can be excellent for improving the reference tracking. This has been proposed in [17] and [16]. The concept is implemented in *ffcon.*
The functions for specialized training have been implemented in a relatively general fashion to allow for model-reference control. That is, the following criterion is minimized:

$$J_2'(\theta) = \sum_t \left(y_m(t) - y(t)\right)^2, \qquad y_m(t) = H(q^{-1})r(t)$$

The user must the specify the desired closed-loop model $H(q^{-1})$.

Input-Output Linearization
Feedback linearization is a common strategy for controlling certain classes of nonlinear processes. The toolkit offers a simple example of a discrete feedback linearizing controller that are based on a neural network model of the process. The NNSYSID toolbox contains a function, which identifies models with the following structure:

$$\hat{y}(t) = f\left(y(t), \ldots, y(t-n+1), u(t-1), \ldots, u(t-m+1)\right) + g\left(y(t), \ldots, y(t-n+1), u(t-1), \ldots, u(t-m+1)\right)u(t)$$

with *f* and *g* being two separate networks. An input-output linearizing controller is obtained by calculating the controls according to:

$$u(t) = \frac{w(t) - f\left(y(t), \ldots, y(t-n+1), u(t-1), \ldots, u(t-m+1)\right)}{g\left(y(t), \ldots, y(t-n+1), u(t-1), \ldots, u(t-m+1)\right)}$$

Selecting the virtual control, *w*, as an appropriate linear combination of past outputs plus the reference allows for an arbitrarily assignment of the closed-loop poles. As for the model-reference controller, feedback-linearization is thus a nonlinear counterpart to pole placement with full zero cancellation (see [17]).

fblcon	Simulate control by feedback linearization.
fblinit	File with design parameters for *fblcon.*

Optimal Control
A simple training algorithm for design of optimal controllers has been implemented by a small modification of the specialized training algorithm. The modification consists of an additional term which are added to the criterion to penalize squared controls:

$$J_3(\theta) = \sum_t \left(r(t) - y(t)\right)^2 + \rho\left(u(t)\right)^2, \qquad \rho \geq 0$$

The training of the network is very similar to specialized training and is also performed on-line. As for specialized training, a "forward" model of the process must be identified in advance with the NNSYSID toolbox in advance. The following files are provided:

opttrain	Train optimal controller with recur. Gauss-Newt.
optrinit	File with design parameters for *opttrain.*
optcon	Simulate optimal control (similar to *invcon*).
optinit	File with design parameters for *optcon.*

Instantaneous Linearization

In [16] a technique for linearizing neural network models around the current operating point is pursued. The idea is summarized in the following. Assume that a deterministic model of the process under consideration has been established with the NNSYSID-toolbox:

$$y(t) = g(y(t-1), \dots, y(t-n), u(t-d), \dots, u(t-d-m))$$

The "state" $z(t)$ is then introduced as a vector composed of the arguments to the function g:

$$z(t) = [y(t-1) \quad \cdots \quad y(t-n) \quad u(t-d) \quad \cdots \quad u(t-d-m)]^T$$

At time $t=\tau$ linearize g around the current state $z(\tau)$ to obtain an approximate model:

$$\tilde{y}(t) = -a_1\tilde{y}(t-1) - \dots - a_n\tilde{y}(t-n) + b_0\tilde{u}(t-d) + \dots + b_m\tilde{u}(t-d-m)$$

where

$$a_i = -\left.\frac{\partial g(z(t))}{\partial y(t-i)}\right|_{z(t)=z(\tau)}$$
$$b_i = \left.\frac{\partial g(z(t))}{\partial u(t-d-i)}\right|_{z(t)=z(\tau)}$$

and

$$\tilde{y}(t-i) = y(t-i) - y(\tau-i)$$
$$\tilde{u}(t-i) = u(t-i) - u(\tau-i)$$

Seperating the portion of the expression containing components of the current state vector, the approximate model may alternatively be written as:

$$y(t) = (1 - A(q^{-1}))y(t) + q^{-d}B(q^{-1})u(t) + \zeta(\tau)$$

where the *bias* term, $\zeta(\tau)$, is determined by

$$\zeta(\tau) = y(\tau) + a_1 y(\tau-1) + \dots + a_n y(\tau-n) - b_0 u(\tau-d) - \dots - b_m u(\tau-d-m)$$

and

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

The approximate model may thus be interpreted as a linear model affected by a DC-disturbance, $\zeta(\tau)$, depending on the operating point. Clearly, it is straightforward to apply this principle for design of control systems. The control systems that result from this are in some sense gain scheduling controllers with an infinite schedule. As opposed to the previously mentioned concepts, the controller is in this case not directly implemented by a neural network. The following files are associated with this principle:

lincon	Simulate control using approximate pole placement or minimum variance.
lininit	File with design parameters for *lincon.*
diophant	General function for solving Diophantine equations.
dio	Prepares problem for being solved by *diophant.*
apccon	Simulate control using approximate generalized predictive control.
apcinit	File with design parameters for *apccon.*

Approximate pole placement and minimum variance:
Together *lincon*, *lininit*, and *diophant* have adopted this idea for realization of different controllers. Three different concepts have been implemented:

- *Pole placement with all zeros canceled.* The zeros are canceled and the controller is designed to make the closed-loop system follow a desired transfer function model.
- *Pole placement with no zeros canceled.* Only the poles of the closed-loop system are moved to prescribed locations.
- *Minimum Variance.* Based on the assumption that the bias, $\zeta(\tau)$, is integrated white noise: $\zeta(\tau) = \frac{e(t)}{\Delta}$, the so-called *MV1*-controller has been implemented. This controller is designed to minimize:

$$J_4(t,u(t)) = E\left\{\left[y(t+d) - W(q^{-1})r(t)\right]^2 + [\Delta u(t)]^2 \middle| I_t\right\}$$

where I_t specifies the information gathered up to time t:

$$I_t = \{y(t), y(t-1), \dots, y(0), u(t-1), \dots, u(0)\}$$

The functions *dio* and *diophant* are provided for solving the Diophantine equations.

Approximate GPC:
The idea behind generalized predictive control (GPC), which is implemented in *apccon* and *apcinit*, is at each iteration to minimize a criterion of the following type:

$$J_5(t,U(t)) = \sum_{i=N_1}^{N_2}\left[w(t+i) - \hat{y}(t+i)\right]^2 + \rho\sum_{i=1}^{N_u}[\Delta u(t+i-1)]^2$$

with respect to the N_u future controls

$$U(t) = [u(t) \quad \dots \quad u(t+N_u-1)]^T$$

and subject to the constraint

$$\Delta u(t+i) = 0, \qquad N_u \le i \le N_2 - d$$

N_1 is denoted the minimum costing horizon, N_2 the prediction (or maximum costing) horizon, and N_u the (maximum) control horizon. ρ is a weighting factor penalizing variations in the controls. $\zeta(\tau)$, is modelled as integrated white noise and the future predictions are determined as the minimum variance predictions. The optimization problem, which must be solved at each sample, results in a sequence of future controls, $U(t)$. From this sequence the first component, $u(t)$, is then applied to the process. [11] details the concept and describes an application.

Nonlinear Predictive Control

The instantaneous linearization technique has its shortcomings if the nonlinearities are not relatively smooth. Unfortunately practically relevant criterion based design methods founded directly on the nonlinear neural network model are few. One of the most promising strategies is a nonlinear version of the predictive controller discussed above. Only this time the predictions of future outputs are not obtained through a linearization, but from succesive recursion of the nonlinear model:

$$\begin{aligned}\hat{y}(t+k|t) = g\big(&\hat{y}(t+k-1),\ldots,\hat{y}(t+k-\min(k,n)),\\ &y(t),\ldots,y(t-\max(n-k,0)),\\ &u(t-d+k),\ldots,u(t-d-m+k)\big)\end{aligned}$$

The optimization problem is in this case much more difficult to solve and an iterative search method is required. It is referred to [9] for a derivation of the control law and for a discussion of relevant optimization algorithms. The toolkit provides two methods for solving the problem: A Quasi-Newton method applying the BFGS-algorithm for updating the inverse Hessian and full-Newton based Levenberg-Marquardt method.

npccon1	Simulate NPC using a Quasi-Newton method.
npccon2	Simulate NPC using a Newon based Levenberg-Marquardt method.
npcinit	File with design parameters.

4. Perspectives on Real-Time Implementation

The NNCTRL toolkit has been developed to comply with the Integrated Real-time Control and Simulation Tool (IRCST) developed at the Department of Automation, DTU. The design considerations behind the IRCST system and the basic structure of the system are described in this section.

One of the primary objective when designing the IRCST system was to create a tool to make experimental verification more easily available to control algorithm developers. Experimental verification is a very important part of the control system design. However, in many cases it is omitted because of the inherent problems associated with laboratory setups, real-time computing, programming etc. The IRCST system attempts to solve some of these problems for the algorithm developers by offering a number of buildings blocks (i.e., C-functions) and a "template." The template provides a structure for simulation and real-time code for the algorithm designers without enforcing constraints and limiting the use of new algorithms and analysis methods.

One important consideration in the design of IRCST has been the possibility of porting algorithms between different hardware platforms. In the implementation of the IRCST system great care has been taken to ensure the portability of the IRCST system to other real-time platforms than the ones it was originally designed for through a clear distinction between platform dependent and platform independent code. There are not any other tools available with the same functionality, but several tools that address the same problems. Some examples are ControlShell [14], Chimera/Okina [2] and Realtime Workshop [13]. IRCST facilitates datalogging, changing of design parameters and structural changes in the controller code. It supports algorithm portability, provides interfacing to data visualization and analysis tools in MATLAB.

The use of IRCST

The IRCST system consists of three components which can be used independently, but are most useful together:

- A simulation tool/template where the process, e.g., modelled using SIMULINK, can be investigated when different controllers are used. In the IRCST context this is where the NNCTRL toolkit fits in. In the NNCTRL version of the simulation template the proces to be controlled can also be represented by a pre-trained neural network if no other model is available. The control algorithms are programmed as MATLAB scripts and also included. The template is based on a distinction in code and data (*con.m and *init.m files, respectively), meaning that the code normally should not be modified during simulation. If a reconfiguration is desired the corresponding parameter in the data portion is modified. All changes are made in the initialization file while the control system simulation program is left untouched during normal simulations.
- A real-time tool/template, where the control algorithm programmed in C is included. The structure of this template is equivalent to the structure of the simulation template and is also based on the *init/*con concept. The transformation of the algorithm from MATLAB to C is supported by the third component.
- A methodology and a software library of platform independent functions coded in C corresponding to MATLAB functions that can support the transformation of MATLAB code to C code. The transformation of the control algorithm from MATLAB to C is done by the algorithm designer who has complete control over the process. Currently programs that automatically translates MATLAB code to C or C++ code are being developed [8] but none of these can be used at this point. The MATLAB compiler from MathWorks, Inc. is not

useful in this case as it runs only on supported MATLAB platforms, which are normally not real-time platforms.

Fig. 3 shows the relation between the simulation template and the real-time/C template.

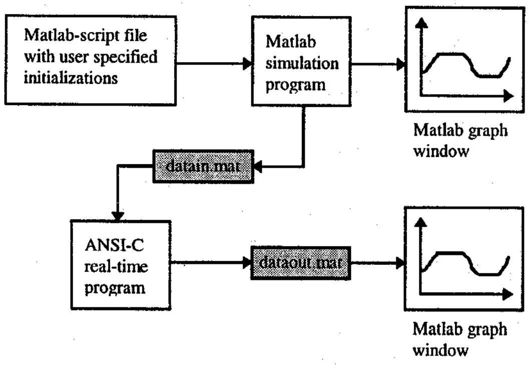

Figure 3. Overall structure of IRCST.

The user develops and tests the control algorithm in The MATLAB/SIMULINK portion of IRCST in the upper part of fig. 4. When the algorithm is performing satisfactorily it is translated to C and build into the real-time template for the relevant real-time hardware platform. From the parameters in the initialization file used during the simulation experiments, a binary MATLAB file is generated for reading the initialized variables into the real-time template. In that way exactly the same data is used for experiment in the laboratory as during the simulations. The real-time template writes user specified variables into binary MATLAB files, which in turn can be read into MATLAB, visualized and analyzed or compared to simulation results. It is simple to test new combinations using simulation before the use in real-time.

The real-time and simulation templates of IRCST are used for different projects and courses at IAU by students as well as by the research staff.

5. Conclusions

A MATLAB toolkit for construction and simulation of neural network based control systems has been presented. The toolkit provides a number of ready-made examples of control systems which can be used directly. The different controllers are implemented in the same basic framework (or "template") which is very easy to modify or extend. Therefore, the user can easily apply of the toolkit for testing his/her own controllers as well.

The toolkit has been developed to comply with IRCST (Integrated Real-time Control and Simulation Tool) to allow for a simple experimental verification of the different control systems.

The NNCTRL toolkit is available from The Institute of Automation WWW site. The address is:
http://www.iau.dtu.dk/Projects/proj/nnctrl.html

References

[1] R. Fletcher, "Practical Methods of Optimization," Wiley, 1987.

[2] M.W. Gertz, P.K. Khosla, "ONIKA Iconic Programming Language and Human-Machine Interface," Dept. of Elec. and Comp. Eng. and the Robotics Institute, Carnigie-Mellon University.

[3] A. Grace, A.J. Laub, J.N Little, C.M. Thompson , "Control System Toolbox User's Guide," The MathWorks, Inc, 1992.

[4] K.J. Hunt, D. Sbarbaro, "Neural Networks for Nonlinear Internal Model Control," IEE Proceedings-D, Vol. 138, No. 5, pp. 431-438, 1991.

[5] K.J. Hunt, D. Sbarbaro, R. Zbikowski, P.J. Gawthrop, "Neural Networks for Control Systems - A Survey," Automatica, Vol. 28, No. 6, pp. 1083-1112, 1992.

[6] W.T. van Luenen, "Neural Networks for Control: on Knowleige Representation and Learning," Ph.D. Thesis, Control Laboratory of Electrical Engineering, University of Twente, Enschede, the Netherlands, 1993.

[7] L. Ljung,"System Identification - Theory for the User," Prentice-Hall, 1987.

[8] MATCOM, "MATCOM - a MATLAB to C++ translator and support libraries," March '96 release, 1996.

[9] M. Nørgaard, P.H. Sørensen, "Generalized Predictive Control of a Nonlinear System using Neural Networks," Preprints, 1995 International Symposium on Artificial Neural Networks, Hsinchu, Taiwan, pp. B1-33-40, 1995.

[10] M. Nørgaard, O. Ravn, L.K. Hansen, N.K. Poulsen, "The NNSYSID Toolbox - A MATLAB Toolbox for System Identification with Neural Networks," accepted for the 1996 IEEE Symposium on Computer-Aided Control System Design, Dearborn, Michigan, USA, 1996.

[11] M. Nørgaard, P.H. Sørensen, N.K. Poulsen, O. Ravn, & L.K. Hansen, "Intelligent Predictive Control of Nonlinear Processes Using Neural Networks," accepted for the 11th IEEE Int. Symp. on Intelligent Control (ISIC), Dearborn, Michigan, USA, 1996.

[12] D. Psaltis, A. Sideris, A.A. Yamamure, "A Multilayered Neural Network Controller," Control Sys. Mag., Vol. 8, No. 2, pp. 17-21, 1988.

[13] *"Real-Time Workshop, User's Guide,"* The MathWorks, Inc, 1994.

[14] S.A. Schneider, V.W. Chen, G. Pardo-Castellote, "ControlShell: a Real-Time Software Framework," proc. of the IEEE Int. Conf. on Robotics and Automation, 1994.

[15] J. Sjöberg, H. Hjalmerson, L. Ljung, "Neural Networks in System Identification," Prep 10th IFAC symp. SYSID, Copenhagen, Denmark. Vol.2, pp. 49-71, 1994.

[16] O. Sørensen,"Neural Networks in Control Applications," Ph.D. Thesis. Aalborg University, Department of Control Engineering, 1994.

[17] K.J. Åström, B. Wittenmark, "Adaptive Control," 2nd. Edition, Addison-Wesley, 1995.