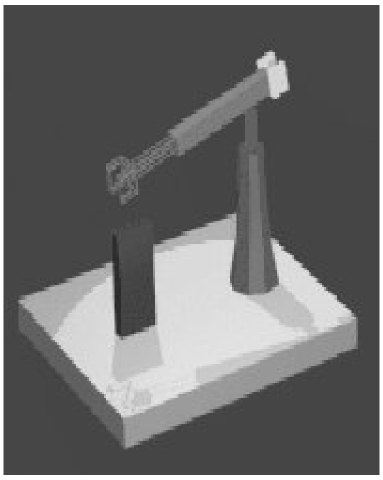
Neural Network Based Control System Design TOOLKIT

Version 2

For Use with MATLAB®



Magnus Nørgaard

Department of Automation
Department of Mathematical Modelling

Technical Report 00-E-892, Department of Automation

Technical University of Denmark

Release Notes

Neural Network Based Control System Design Toolkit

Version 2.0

Department of Automation, Technical University of Denmark, January 23, 2000

This note contains important information on how the present set of tools is to be installed and the conditions under which it may be used. Please read it carefully before use.

It is important that version 2.0 of the NNSYSID toolbox (Neural Network based SYStem IDentification) has been installed in advance.

INSTALLING THE TOOLKIT

° The present version of the toolkit (version 2) is provided for use with MATLAB 5.3 or higher. It has been tested under Windows NT/98, Linux, and HP-UX.

° The entire toolkit is implemented as ordinary m-files and thus it should work equally well on all hardware platforms. A fast computer is, however, highly recommended.

° No "official" MathWorks toolboxes are necessary for being able to use NNCTRL, but the Control System Toolbox is used in one of the demonstration programs ("lintest"). Although not a requirement, it is definitely an advantage if SIMULINK is available.

° When properly installed, the structure of the toolkit is as follows:

- ***NNCTRL***
  Basic directory containing different Readme-files and the following three subdirectories:

  - ***CTRLTOOL***
    The actual toolkit functions and script-files.

  - ***CTRLDEMO***
    Initialization files, SIMULINK models, and mat-files used for demonstration.

  - ***TEMPLATE***
    "Templates" for the initialization files which are called by the programs in the *CTRLTOOL* directory.

Your MATLAB path must include the directory *CTRLTOOL* as well as the directory containing the NNSYSID toolbox:

>> *path(path,'/xx/xx/NNCTRL20/CTRLTOOL')*
>> *path(path,'/xx/xx/NNSYSID20')*

If the tools are going to be used on a regular basis it is recommended that the path statements are included in ones personal *startup.m* file (see the manual for MATLAB).

During normal use one begins by copying the initialization file associated with the desired control system from the *TEMPLATE* directory to the working directory. The file must then be modified to comply with the application under consideration. Typical working procedures can be seen by running the demonstration programs. Furthermore, the different text files found in the *NNCTRL20* directory provide supplementary information on this matter.

When running the demonstration programs the working directory must be the directory *NNCTRL20/CTRLDEMO*.

° The checks for incorrect program/function calls are not very thorough and consequently MATLAB will often respond with quite incomprehensible error messages when a program or function is incorrectly invoked.

CONDITIONS/ DISCLAIMER

Magnus Nørgaard
Department of Automation, Building 326
Technical University of Denmark
2800 Kgs. Lyngby
Denmark
e-mail:toolbox@magnusnorgaard.dk

1 Tutorial

This manual documents an engineering tool for design and simulation of control systems for processes that are nonlinear and difficult to model in a deductive fashion. The approach to the problem has been the typical system identification approach to model based control of an unknown process:

1. System identification. That is, to infer a neural network model of the process to be controlled from a set of data collected in an experiment with the process.

2. Based on the identified model, one or more controllers are designed (these may be neural networks as well). Run some simulations to tune the design parameters and select the controller that appears to be most suitable for the application.

3. Implementation on a real-time platform and application to the real process.

In practice the three stages are not necessarily completely independent, and the controller is designed in a more iterative fashion.

While the NNSYSID toolbox described in Nørgaard (2000) was explicitly designed for solving the system identification task, the NNCTRL toolkit has been developed to assist the control engineer in solving the second task. The toolkit is developed in MATLAB due to the excellent data visualization features and its support for simulation of dynamic systems. In addition, it has been a major motivation that MATLAB is extremely popular in the control engineering community. Apart from the NNSYSID toolbox it is also an advantage if SIMULINK® is available. If this is not present, MATLAB's built-in ODE solver is used instead.

The NNCTRL toolkit provides concepts where a network is used directly as the controller *as well as* indirect designs that are based on a neural network process model. The concepts supported by the toolkit include: Direct inverse control, internal model control, feedforward control, optimal control, feedback linearization, nonlinear generalized predictive control, and control based on instantaneous linearization. The toolkit is intended to be used primarily on time-invariant, single-input, single-output (SISO) processes.

The NNCTRL toolkit has been given a flexible structure to accommodate incorporation of the user's personal neural network architectures and control system designs. Since the entire toolkit has been implemented as ordinary 'm-files' it is very easy to understand and modify the existing code if desired. This is an attractive feature if the designs provided in advance are not considered sufficient for the applications under consideration or if the user would like to test new ideas.

First the manual describes the fundamental program structure shared by the different neural network based control systems and training algorithms. Subsequently an example is given of how the user specifies the design parameters associated with a given application. Finally the different functions in the toolkit are presented by category in accordance with the type of control system to which they belong.

1.1 THE BASIC FRAMEWORK

All the different control systems have been implemented within the same framework. This framework is composed of different standard elements such as reading a design parameter file, variable initialization, generating a reference signal, process simulation, a constant gain PID controller, data storage, data visualization, and more. The control systems can be divided into two fundamentally different categories:

- Direct design: the controller *is* a neural network.
- Indirect design: The controller is not itself a neural network, but it is based on a neural network model of the process.

The program structure is slightly different for each of the two categories. To illustrate the difference they are shown in fig. 1 and fig. 2, respectively.

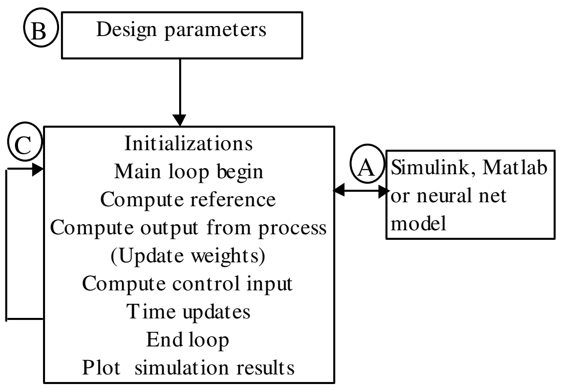


Figure 1. *Program structure for the direct design (i.e., the controller is a neural network). In some cases it is possible to train the controller on-line and for this reason a step called: "update weights" has been included.*

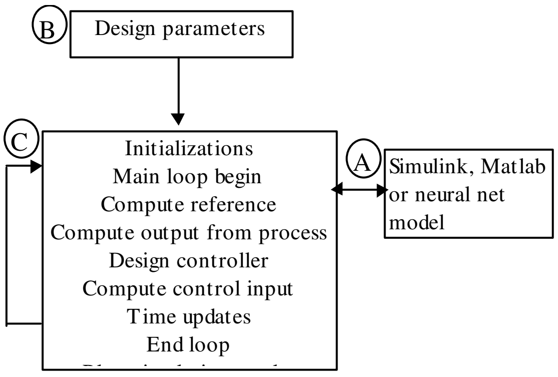


Figure 2. *Program structure for the indirect design, i.e., when the controller is based on a neural network model of the process.*

Each of the boxes in Fig. 1 and Fig. 2 specifies a MATLAB script file or function. The three basic components are:

A) A function describing the process to be controlled. The process can be specified as 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 relevant only when a mathematical model of the process is available in advance.

B) A MATLAB script file containing design parameters and variables to be initialized by the user. The initializations that are typically required include: 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 that is associated with the type of control system. The name is always concluded by the letters *init.m* (e.g., *invinit.m* in direct inverse control and *npcinit.m* in nonlinear predictive control). A typical NNCTRL session is initiated by copying a “template” initialization file to the working directory. This file is then modified to comply with the application under consideration.

C) 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 specify that it is the control system simulation program (for example *invcon.m* in direct inverse control and *optcon.m* in optimal control). It should be emphasized that the program structure does not always follow the patterns shown in fig. 1 and fig. 2 exactly, but that small variations may occur.

1.2 A TYPICAL INITIALIZATION FILE

During normal operation the user will only need to modify an initialization file containing the basic design parameters and initializations for the simulation/training. More experienced users may also wish to modify the actual simulation programs to incorporate their personal algorithms, but this topic is not covered here. As an example of a typical initialization file, the file used for simulation of nonlinear predictive control is shown on the following page. Subsequently the different design parameters are discussed.

```
% -------------------------------> NPCINIT.M <------------------------------

% ----------      Switches       -----------
regty      ='npc';            % Controller type (npc, pid, none)
refty      ='siggener';       % Reference signal (siggener/<var. name>)
simul      ='simulink';       % Control object spec. (simulink/matlab/nnet)


% ----------   Initializations   -----------
Ts = 0.2;                     % Sampling period (in seconds)
samples = 300;                % Number of samples in simulation
u_0 = 0;                      % Initial control input
y_0 = 0;                      % Initial output
ulim_min = -Inf;              % Minimum control input
ulim_max = Inf;               % Maximum control input


% --  System to be Controlled (SIMULINK) -
integrator= 'ode45';          % Name of dif. eq. solver (f. ex. ode45 or ode15s)
sim_model = 'spm1';           % Name of SIMULINK model

% ---  System to be Controlled (MATLAB)  --
mat_model = 'springm';        % Name of MATLAB model
model_out = 'smout';          % Output equation (function of the states)
x0        = [0;0];            % Initial states


% ----- Neural Network Specification ------
% "nnfile" must contain the following variables which together define
% an NNARX model:
% NN, NetDef, W1, W2
% (i.e. regressor structure, architecture definition, and weight matrices)
nnfile  = 'forward';          % Name of file containing the neural network model


% ------------ Reference filter ---------------
Am = [1.0 -0.7];              % Denominator of filter
Bm = [0 0.3];                 % Numerator of filter


% ---------- GPC initializations -----------
N1 = 1;                       % Minimum output horizon (must be equal to time delay!)
N2 = 7;                       % Maximum output horizon (>= nb)
Nu = 2;                       % Control horizon
rho = 0.03;                   % Penalty factor on differenced control signal

% -- Minimization algorithm initialzations -
% maxiter: Maxiumum number of Levenberg-Marquardt/Quasi-Newton iteratons.
% delta  : If the 2-norm of the difference to the previous iterate is smaller
%          than 'delta', the algorithm is terminated.
% initval: A string which will be evaluated by npccon1/2. To avoid problems with
%          local minima, different initial values of the most future component
%          of the 'sequence of future controls' can be initialized differenly.
%          I.e. the minimization algorithm is executed once for each element
%          in 'initval'. Use initval = '[upmin(Nu)]' as default.
maxiter = 5;                  % Max. number of iterations to determine u
delta = 1e-4;                 % Norm-of-control-change-stop criterion
initval = '[upmin(Nu)]';


% ----------- Reference signal -------------
dc      = 0;                  % DC-level
sq_amp  = 3;                  % Amplitude of square signals (row vector)
sq_freq = 0.1;                % Frequencies of square signals (column vector)
sin_amp = [0];                % Amplitude of sine signals (row vector)
sin_freq= [0]';               % Frequencies of sine signals (column vector)
Nvar  = 0';                   % Variance of white noise signal


% -- Constant Gain Controller Parameters --
K=8;                          % PID parameters
Td=0.8;                       % PID
alf=0.1;                      % PID
Wi=0.2;                       % PID (1/Ti)


% ------ Specify data vectors to plot  -------
% plot_a and plot_b must be cell structs containing the vector names in strings
plot_a = {'ref_data','y_data'};
plot_b = {'u_data'};
```

```
% ----------      Switches       -----------
regty      ='npc';        % Controller type (npc, pid, none)
refty      ='siggener';   % Reference signal(siggener/<var.name>)
simul      ='simulink';   % Control object spec.(simulink/matlab/nnet)
```

`regty` defines the type of controller. It is possible to make a complete open-loop simulation (`'none'`), to use a constant gain PID-controller (`'pid'`) or, in this case, a generalized predictive control design based on a neural network predictor for the process (`'npc'`).

`refty` defines the type of reference trajectory. It is possible to use a signal generator (`'siggener'`) which can produce square waves, sinusoidals and white noise. Alternatively one can specify the name of a vector containing the desired reference trajectory (for example, `refty='myref'`). If the latter option is used, the vector must exist in the workspace or be defined elsewhere in the initialization file.

`simul` specifies the "tool" used for modelling the process. There are three possibilities: a SIMULINK model (`'simulink'`), a MATLAB model (`'matlab'`), or a neural network model (`'nnet'`).

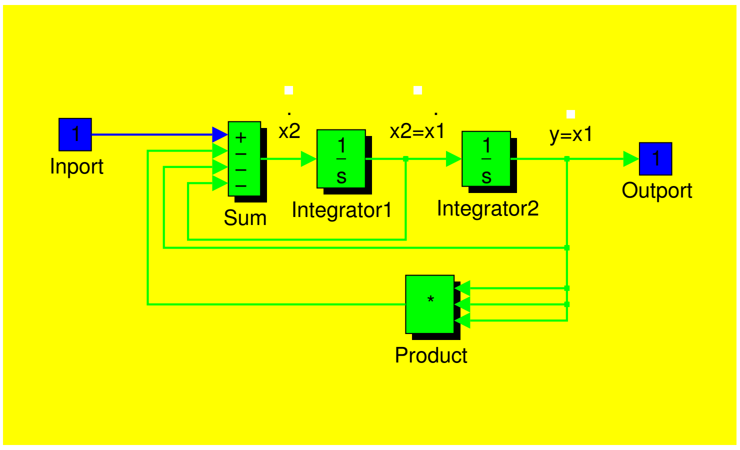
```
% ----------   Initializations   -----------
Ts = 0.2;                    % Sampling period (in seconds)
samples = 300;               % Number of samples in simulation
u_0 = 0;                     % Initial control input
y_0 = 0;                     % Initial output
ulim_min = -Inf;             % Minimum control input
ulim_max = Inf;              % Maximum control input
```

`Ts` is the sampling period in seconds, and `samples` specifies the number of samples to be simulated. The control input and system output corresponding to the initial state can be specified in the variables `u_0` and `y_0`. `ulim_min and ulim_max` can be used for imposing constraints on the control input.

```
% --  System to be Controlled (SIMULINK) --
integrator= 'ode45';         % Name of dif. eq. solver (f. ex. ode45 or ode15s)
sim_model = 'spm1';          % Name of SIMULINK model
```

If `simul='simulink'` the program will simulate the "process" specified in a SIMULINK diagram with the name defined in `sim_model`. The SIMULINK block must have an *inport* specifying the input and an *outport* specifying the output, but apart from this there are no constraints on how the model is built. The variable *integrator* contains the name of the ODE solver. *'ode45'* is the one used most often. The reader is referred to the MATLAB manuals for more details on the different ODE solvers.

The picture below shows the SIMULINK model used in the demos.



```
% ---  System to be Controlled (MATLAB)  --
mat_model = 'springm';       % Name of MATLAB model
model_out = 'smout';         % Output equation (function of the states)
x0        = [0;0];           % Initial states
```

If `simul='matlab'` the program will simulate the "process" defined by the two MATLAB files defined in `mat_model` (the differential equations) and `model_out` (the output as a function of the states).

For example:

```
function xdot=springm(t,x)
global ugl;
xdot = [x(2) ; -x(1)-x(2)-x(1)*x(1)*x(1)+ugl];
```

and:

```
function y=smout(x);
y = x(1);
```

x0 is a vector specifying the initial states.

It is important to include the `global ugl;` statement (`ugl` is short for "u global") in the MATLAB function containing the differential equations describing the model. This is to comply with the format required for passing the control signal to a function called by MATLAB's differential equation solver *ode45*. The 'ode45' solver has been pre-selected. If the user prefers another ODE solver, the name of this should be inserted in the main program.

```
% ----- Neural Network Specification ------
% "nnfile" must contain the following variables which together define
% an NNARX model:
% NN, NetDef, W1, W2
% (i.e. regressor structure, architecture definition, and weight matrices)
nnfile  = 'forward';       % Name of file containing the neural network model
```

The predictive controller requires a neural network model to predict the future outputs. This network must be a NNARX (or NNOE) model created with the NNSYSID-toolbox. Regressor structure (*NN*), network architecture (*NetDef*) and weight matrices (*W1* and *W2*) must be saved in a file. The name of the file name must be specified in the variable *nnfile*. If neither a SIMULINK model nor a MATLAB model of the process exists, the network model can also be used for simulating the process.

```
% ------------ Reference filter ---------------
Am = [1.0 -0.7];              % Denominator of filter
Bm = [0.3];                   % Numerator of filter
```

It is possible to filter the reference trajectory if desired. The filter's transfer function is specified in the polynomials A_m and B_m. The specified choice corresponds to:

$$r(t) = \frac{0.3}{1-0.7q^{-1}}\bar{r}(t)$$

```
% ---------- GPC initializations -----------
N1 = 1;               % Min. prediction horizon (must equal the time delay!)
N2 = 7;               % Max. prediction horizon (>= nb)
Nu = 2;               % Control horizon
rho = 0.03;           % Penalty factor on differenced control signal
```

```
%-- Minimization algorithm initialzations -
%maxiter: Maxiumum number of Levenberg-Marquardt/Quasi-Newton iteratons.
%delta  : If the 2-norm of the difference to the previous iterate is smaller
%         than 'delta', the algorithm is terminated.
%initval: A string which will be evaluated by npccon1/2. To avoid problems with
%         local minima, different initial values of the most future component
%         of the 'sequence of future controls' can be initialized differenly.
%          I.e. the minimization algorithm is executed once for each element
%          in 'initval'. Use initval = '[upmin(Nu)]' as default.
maxiter = 5;                 % Max. number of iterations to determine u
delta = 1e-4;                % Norm-of-control-change-stop criterion
initval = '[upmin(Nu)]';
```

If the nonlinear generalized predictive control strategy has been selected by setting `regty='npc'` the design parameters must be specified in `N1`, `N2`, `Nu`, and `rho`:

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

The GPC-criterion can be minimized using two different schemes: a Quasi-Newton algorithm (obtained by running the program *npccon1)* or a Levenberg-Marquardt algorithm (obtained by running *npccon2). `maxiter`* specifies the maximum number of iterations and `delta` defines the stopping criterion. The criterion may have several local minima and consequently it may be desireable to specify different starting points for the minimization algorithm. This can be done in the vector `initval`.

```
% ----------- Reference signal -------------
dc      = 0;                 % DC-level
sq_amp  = [3];               % Amplitude of square signals (row vector)
sq_freq = [0.1]';            % Frequencies of square signals (column vector)
sin_amp = [0];               % Amplitude of sine signals (row vector)
sin_freq= [0]';              % Frequencies of sine signals (column vector)
Nvar  = 0';                  % Variance of white noise signal
```

If `refty='siggener'` a simple signal generator is invoked to generate the reference trajectory. The variables above define the trajectory. It is possible to create a trajectory composed of several square waves and/or sinusoidals by defining `sq_amp`, `sq_freq`, `sin_amp`, and `sin_freq` as vectors.

```
% -- Constant Gain Controller Parameters --
K=8;                         % PID parameters
Td=0.8;                      % PID
alf=0.1;                     % PID
Wi=0.2;                      % PID (1/Ti)
```

It is possible to use an ordinary PID controller instead of the predictive controller. This is achieved by setting `regty='pid'`. The controller is implemented as a discrete version of:

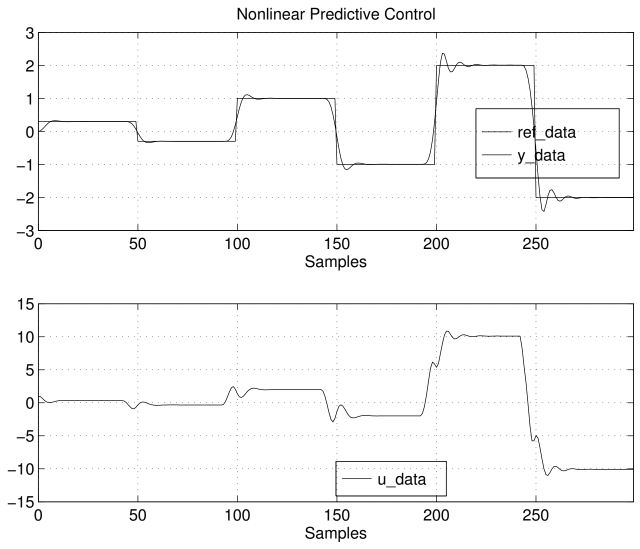
$$D(s) = K\frac{1+T_d s}{1+alf*T_d s}\frac{1+T_i s}{T_i s}$$

```
% ------ Specify data vectors to plot  -------
% plot_a and plot_b must be cell structures containing the vector names in strings
plot_a = {'ref_data','y_data'};
plot_b = {'u_data'};
```

The final lines in the file specify the signals to be plotted. The signals considered most interesting are stored in vectors with names concluded by the letters '*_data*'. For example, *ref_data* contains the reference, *y_data* the output signal and *u_data* the control signal. *plot_a* and *plot_b* are cell structures containing, in each element, a sting which is evaluated by the program. For example, it is possible to write:

```
plot_b = {'3.5*u_data+10'};
```

to plot a scaled version of the control signal. Since the simulation programs are script files, all parameters, vectors and matrices will be available in the workspace after the simulation, and if necessary they can be analysed further. A typical plot produced by the simulation program is shown below.


Nonlinear Predictive Control
ref_data
y_data
Samples
u_data
Samples

2 Control system design

In this chapter the control systems implemented in the NNCTRL toolkit are introduced. The control systems provided are: direct inverse control, internal model control, feedforward, control by input-output linearization, optimal control, approximate pole placement control, minimum variance control, predictive control based on instantaneous linearization, and nonlinear predictive control.

To experience the features of the different types of neural network based control it is often relevant to investigate how the controllers perform on a well-known process. In fact, this is one of the primary purposes of this toolkit. For this reason it is possible to control models that are built in SIMULINK® or differential equation models that are specified in MATLAB®. Also it is possible to *simulate* an experiment with the process to acquire a set of data for system identification and controller design.

In the following sections the different functions are discussed, starting with how to simulate an experiment (if externally generated data are used, one can skip this section). Not all the options for the different *init.m files will be discussed since they have a great deal of overlap with one another and with the one shown in the previous chapter.

2.1 SIMULATING THE EXPERIMENT

Experiment Simulation	
experim	Simulate experiment.
expinit	File with design parameters for *experim.*

If the NNCTRL toolkit is used to gain experience with neural network based control or if a rudimentary model of the process to be controlled exists, the experiment can be simulated using a SIMULINK® or MATLAB® model of the process. The experiment can be conducted in open-loop, but if the process is poorly damped or unstable it is also possible to make the experiment in closed-loop with a PID-controller (or an RST-controller).

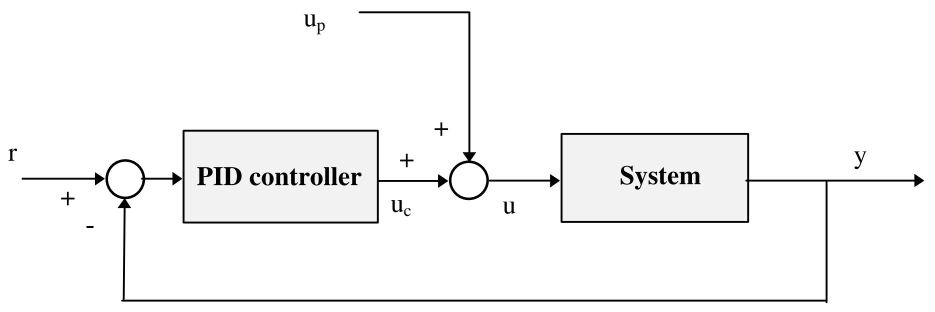


Figure 3 *Collecting data in closed-loop with a PID-controller.*

A manually tuned PID-controller will often be designed somewhat conservatively - its primary objective is to stabilize the process. It is therefore possible to add a signal (u_p) directly to the control signal to ensure a certain high frequency content in the data set. If the experiment is conducted in open-loop it makes no difference whether r or u_p is used as the reference signal.

The function *experim* simulates the experiment and the file *expinit* contains the initializations to be specified by the user. Some excerpts from the *expinit* file are given below:

```
% ----------      Switches       -----------
regty      ='none';          % Controller type (rst, pid, none)
refty      ='predef';        % Reference signal (siggener/none/<var. name>)
probety    ='none';          % Probing signal (none/<var. name>)
```

`regty` specifies the type of controller. `regty='none'` implies that the experiment is conducted in open-loop, `regty='pid'` that the experiment is conducted in closed-loop with a PID-controller, and `regty='rst'` that the experiment is conducted in closed-loop with an RST-controller. The design parameters of the PID and RST controllers are also set in the file:

```
% --------  Linear Controller Parameters  ---------
K=8;                         % PID parameters
Td=0.8;                      % PID
alf=0.1;                     % PID
Wi=0.2;                      % PID (1/Ti)

r0=0.0609;                   % RST parameters
r1=0.0514;                   % RST
t0=0.2;                      % RST
s0=0.802;                    % RST
s1=-0.602;                   % RST
R = [r0 r1];
S = [s0 s1];
T = t0;
```

`refty` selects the type of reference signal (r) and `probety` the type of signal added directly to the control signal in a closed-loop experiment (u_p). If `refty='siggener'` the built-in signal generator is used. If this is not adequate it is possible to use a signal defined in a vector which is available in the workspace. If the name of the vector is `my_signal` then set `refty= 'my_signal'`.

2.2 CONTROL WITH INVERSE MODELS

Control with Inverse Models

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 an RPLR type Gauss-Newton method.
special3	Specialized training with an RPEM type Gauss-Newton method.
invinit2	File with design parameters for *special2* and *special3*.
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 process with feedforward control.
ffinit	File with design parameters for *ffcon*.
invtest	Program for demonstrating direct inverse control.

Conceptually, the most fundamental neural network based controllers are probably those using the "inverse" of the process as the controller. The most simple concept is called *direct inverse control.* The principle of this is that if the process can be described by:

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

a network is trained as the inverse of the process:

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

The inverse model is subsequently applied as the 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 available which use this idea, e.g., Psaltis et al. (1988), Hunt & Sbarbaro (1991), and Hunt et al. (1992). See also fig. 4. Internal model control (IMC) and feedforward control represent other strategies that utilize inverse models.

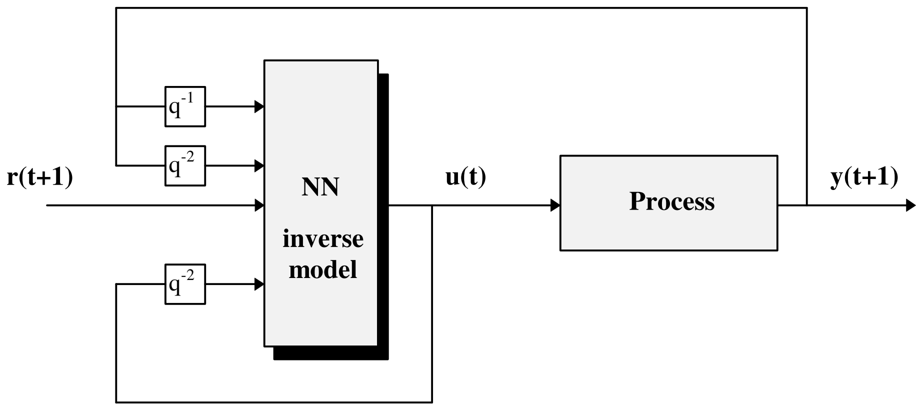


Figure 4 *Direct inverse control.*

Before considering the actual control system, an inverse model must be trained. Two strategies for obtaining the inverse model are provided in the toolkit: generalized training and specialized training (Psaltis et al. 1988). In *generalized training* a network is trained *off-line* to minimize the following criterion (θ specifies the weights in the network):

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

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 (Fletcher, 1987), is invoked.

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

$$J_2(\theta) = \sum_{t} \left(y_m(t) - y(t)\right)^2$$

where

$$y_m(t) = \frac{q^{-1}B_m(q)}{A_m(q)} r(t)$$

The inverse model is obtained if A_m=B_m=1, but often a low-pass filtered version is preferred. In this case the result will be some kind of “detuned” (or “smoothed”) inverse model.

Specialized training is often said to be *goal directed* because it, as opposed to generalized training, attempts to train the network so that the output of the process follows the reference closely. For this reason, specialized training is particularly well-suited for optimizing the controller for a prescribed reference trajectory. This is a relevant feature in many robotics applications. Also it is with special training possible to make an inverse model for processes that are not one-to-one.

Specialized training must be performed on-line and thus it is much more difficult to carry out in practice than generalized training. Before the actual training of the inverse model is initiated, a “forward” model of the process must be trained since this is required by the scheme. This can be created with the NNSYSID toolbox from a data set collected in an experiment performed in advance as described in section 2.1. The principle is depicted in fig. 5.

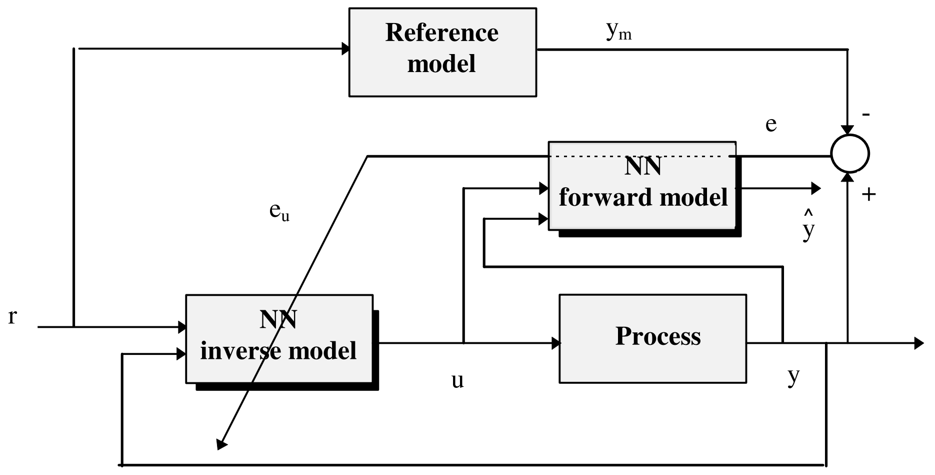


Figure 5 *The principle of specialized training.*

Unlike generalized training the controller design is model based when the specialized training scheme is applied since a model of the process is required. Details on the principle can be found in Hunt & Sbarbaro (1991). Three different variations of the scheme have been implemented: One using a recursive back-propagation algorithm for minimizing the criterion (*special1*) and two more rapid methods using different variations of the recursive Gauss-Newton algorithm (*special2* and *special3).* Specialized training is more complex to implement than generalized training and requires more design parameters. For this reason it has not been implemented as MATLAB functions like generalized training. The three variations of the scheme are instead implemented by the same type of combination of script files and initialization files as was discussed in chapter 1.

The difference between *special 1* and *special2* on one side and *special3* on the other side lies in the way the derivative of the process output with respect to the weights of the inverse model is approximated. *special 1* and *special2* have been implemented in the spirit of recursive "pseudo-linear regression" methods (Ljung, 1987) in the sense that the past outputs and past controls dependency on the network weights is disregarded:

$$\psi(t) = \frac{dy(t)}{d\theta} = \frac{\partial y(t)}{\partial u(t-1)}\frac{du(t-1)}{d\theta} \approx \frac{\partial \hat{y}(t)}{\partial u(t-1)}\frac{\partial u(t-1)}{\partial \theta}$$

special3 is closer to the true recursive prediction error method discussed in Ljung (1987):

$$\psi(t) = \frac{dy(t)}{d\theta} = \frac{\partial y(t)}{\partial u(t-1)}\frac{du(t-1)}{d\theta}$$

$$\approx \frac{\partial \hat{y}(t)}{\partial u(t-1)}\left[\frac{\partial u(t-1)}{\partial \theta} + \sum_{i=1}^{n}\frac{\partial u(t-1)}{\partial y(t-i)}\frac{d\hat{y}(t-i)}{d\theta} + \sum_{i=2}^{m}\frac{\partial u(t-1)}{\partial u(t-i)}\frac{du(t-i)}{d\theta}\right]$$

In *special2* and *special3* the user can choose one of three updating formulas (see Åström & Wittenmark, 1995; Salgado et al, 1988):

Exponential forgetting (*method*=*'ff'*):

$$K(t) = P(t-1)\psi(t)\left(\lambda I + \psi^T(t)P(t-1)\psi(t)\right)^{-1}$$

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\overline{P}(t) = \left(P(t-1) - K(t)\psi^T(t)P(t-1)\right) \Big/ \lambda$$

Constant-trace (*method*=*'ct'*):

$$K(t) = P(t-1)\psi(t)\left(\lambda + \psi^T(t)P(t-1)\psi(t)\right)^{-1}$$

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\overline{P}(t) = \left(P(t-1) + \frac{P(t-1)\psi(t)\psi^T(t)P(t-1)}{1 + \psi^T(t)P(t-1)\psi(t)}\right)^{-1} \Big/ \lambda$$

$$P(t) = \frac{\alpha_{\max} - \alpha_{\min}}{tr(\overline{P}(t))}\overline{P}(t) + \alpha_{\min} I$$

Exponential Forgetting and Resetting Algorithm (*method*=*'efra'*):

$$K(t) = \alpha P(t-1)\psi(t)\left(1 + \psi^T(t)P(t-1)\psi(t)\right)^{-1}$$

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$P(t) = \frac{1}{\lambda}P(t-1) - K(t)\psi^T(t)P(t-1) + \beta I - \delta P^2(t-1)$$

To illustrate how specialized training is executed, some excerpts from the file *invinit2* are shown below. This file contains the design parameters for both *special2* and *special3*. The file *invinit1*, which contains the design parameters for *special1*, has almost the same structure.

```
% ----------       Switches        -----------
simul      = 'simulink';     % System specification (simulink/matlab/nnet)
method     = 'ff';           % Training algorithm (ff/ct/efra)
refty      = 'siggener';     % Reference signal (siggener/<var. name>)


% ------    General Initializations  -------
Ts = 0.20;                   % Sampling period (in seconds)
samples = 200 ;              % Number of samples in each epoch
```

`method` specifies the training algorithm. The forgetting factor algorithm will usually give the most rapid convergence, but be aware of the possibility of covariance blow-up (Åström & Wittenmark, 1995).

`refty` specifies the type of reference signal.

```
% ----- Neural Network Specification ------
%The "forward model file" must contain the following variables which together
% define a NNARX-model:
% NN, NetDeff, W1f, W2f
% and the "inverse model file" must contain
% NN, NetDefi, W1i, W2i
% (i.e. regressor structure, architecture definition, and weight matrices)
nnforw = 'forward';          % Name of file containing forward model
nninv  = 'inverse';          % Name of file containing inverse model
```

`nnforw` should be set to the name of the file containing the architecture definition and weight matrices for a neural network model of the process to be controlled. This model is required to provide estimates of the process' Jacobian: $\frac{\partial \hat{y}(t)}{\partial u(t-1)}$

`nninv` should be set to the name of a file containing the architecture definition and the *initial weight matrices* for the inverse neural network model. The weights can be initialized by random, but most often it is better to make a rough initialization using generalized training. When working in this way, specialized training can be considered a *fine-tuning* of the inverse model.

```
% ----------- Reference Model ---------------
Am = [1 0.7];               % Model denominator
Bm = [0.3];                 % Model numerator
```

When the ordinary inverse model is needed `Am=1` and `Bm=1`. However, often it is preferred that the closed-loop system follows a prescribed transfer function model. `Am` and `Bm` are then be set to the desired closed-loop denominator and numerator, respectively.

```
% ----------- Training parameters -----------
maxiter = 8;
```

`maxiter` specifies the number of times that the reference trajectory (of the length specified in the variable `samples`) is repeated. The total number of times that the weights are updated are thus `maxiter*samples`.

```
% --- Forgetting factor algorithm (ff) ---
% trparms = [lambda p0]
%    lambda = forgetting factor (suggested value 0.995)
%    p0     = Covariance matrix diagonal (1-10)
%
% --- Constant trace algorithm (ct) ---
% trparms = [lambda alpha_max alpha_min]
%    lambda = forgetting factor (suggested value 0.995)
%    alpha_max = Max. eigenvalue of covariance matrix (100)
%    alpha_min = Min. eigenvaule of covariance matrix (0.001)
%
% --- Exponential Forgetting and Restting Algorithm (efra) ---
% trparms = [alpha beta delta lambda]
%    Suggested values:
%    alpha = 0.5-1
%    beta = 0.001
%    delta = 0.001
%    lambda = 0.98
trparms = [0.995 10];
%trparms = [0.995 100 0.001];
%trparms = [1 0.001 0.001 0.98];
```

`trparms` is a vector containing the design parameters for the update of the covariance matrix in the recursive Gauss-Newton algorithm. The content of this vector depends of course on the selected updating scheme (exponential forgetting, constant trace, EFRA).

```
% ------ Specify data vectors to plot  -------
% plot_a and plot_b must be cell structures containing the vector names in strings
plot_a = {'ref_data','y_data','ym_data','yhat_data'};
plot_b = {'u_data'};
```

Each time the entire reference trajectory has been applied, a plot of the signals defined in `plot_a` and `plot_b` will appear on the screen. To give the user an indication of how the algorithm is converging, the SSE (sum-of-squared-errors) between reference and output signal is displayed in the plot.

Once an inverse model has been trained there are different ways in which it can be used for control. The toolkit provides three different concepts:

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

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

The principle was shown in fig. 4.

invcon is used for simulating the closed-loop system while the design parameters are specified in the file *invinit*. Some examples of design parameters to be initialized in *invinit* are shown below:

```
% ----------       Switches       -----------
```

```
regty      ='dic';          % Controller type (dic, pid, none)
simul      ='simulink';     % Control object spec. (simulink/matlab/nnet)
```

`regty='dic'` means *direct inverse control*. If `regty='pid'` an ordinary PID-controller will be used instead.

```
% ----- Inverse Network Specification ------
% The file must contain the variables:
% NN, NetDefi, W1i, W2i
% (i.e. regressor structure, architecture definition, and weight matrices)
nninv = 'inverse';          % Name of file
```

The variable `nninv` is set to the name of the file containing the regressor structure, architecture definition and weight matrices for the neural network that is modelling the inverse of the process.

```
% ------------ Reference Filter ---------------
Am = [1];                   % Filter denominator
Bm = [1];                   % Filter numerator
```

`Am` and `Bm` defines a filter that is used on the reference signal before it is applied to the inverse model. It has *nothing* to do with the *reference model* used in specialized training. An option to filter the reference is provided to avoid that the reference signal has a large high frequency component.

Internal model control:
This design has been treated in Hunt & Sbarbaro (1991). The control signal is synthesized by a combination of a “forward” model of the process and an inverse model. An attractive property of this design is that it provides an off-set free response even if the process is affected by a constant disturbance. The scheme is implemented in *imccon* and *imcinit*.

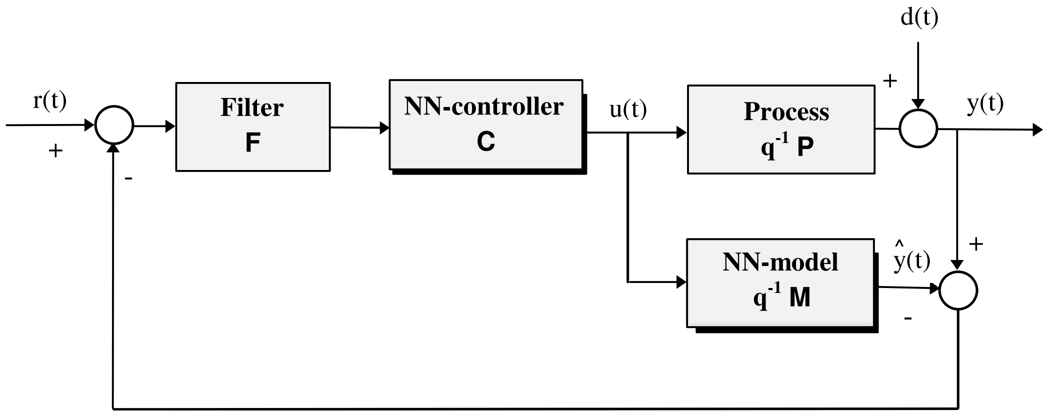


Figure 6 *Inverse model control (IMC).*

The file *imcinit* is very similar to *invinit*. The differences are shown below:

```
% ----------      Switches       -----------
regty      ='imc';          % Controller type (imc, pid, none)
```

`regty` must be set to '`imc`' to select internal model control.

```
% ----- Inverse Network Specification ------
% The file must contain the variables:
% NN, NetDefi, W1i, W2i
% (i.e. regressor structure, architecture definition, and weight matrices)
nninv = 'inverse3';          % File name

% ----- Forward Network Specification ------
% The file must contain: NN, NetDeff, W1f, W2f
nnforw = 'forward';          % File name
```

A neural network model of the process is integrated in the controller design as shown on Fig. 6. If the specialized training scheme was used for creating the inverse model, a model of the process is already available. Otherwise a new model must be trained before the IMC concept can be used.

```
% ---------------- Filter ------------------
Am = [1 -0.7];               % Filter denominator
Bm = [0.3];                  % Filter numerator
```

`Am` and `Bm` are denominator and numerator of the filter, *F*, shown in fig.6.

Feedforward
Using inverse models for feedback control leads to a dead-beat type of control, which is unsuitable in many cases. If a PID controller has already been tuned for stabilizing the process, an inverse model can be used for providing a feedforward signal directly from the reference. The scheme is implemented in *ffcon* and *ffinit*.

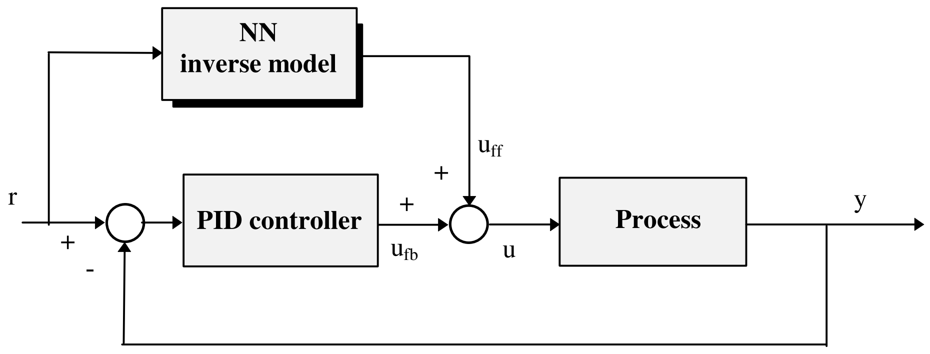


Figure 7 *Feedforward for optimization of an existing control system.*

An excerpt from the file *ffinit* is shown below:

```
% ----------      Switches       -----------
regty      ='pidff';         % Controller type (pid, pidff, ff, none)

% --------  Linear Controller Parameters  ---------
K=8;                         % PID parameters
Td=0.8;                      % PID
alf=0.1;                     % PID
Wi=0.2;                      % PID (1/Ti)
```

If `regty='pid'` a constant gain PID controller is used with the parameters defined by `K`, `Td`, `alf`, and `Wi`. If `regty='ff'` the process is controlled in open-loop using the inverse neural network model as a feedforward controller. `regty='pidff'` combines the two so that the PID controller is used for stabilizing the process and suppressing disturbances while the feedforward is used for providing a fast tracking of the reference.

2.3 FEEDBACK LINEARIZATION

Control by Feedback Linearization

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

Feedback linearization is a common method for controlling certain classes of nonlinear processes. The toolkit provides a simple example of a discrete input-output linearizing controller that is based on a neural network model of the process. The NNSYSID toolbox contains the function *nniol* to train a neural network model that has the following structure:

$$\begin{aligned}\hat{y}(t) = f\big(&y(t-1),\dots,y(t-n),u(t-2),\dots,u(t-m)\big)+\\ &g\big(y(t-1),\dots,y(t-n),u(t-2),\dots,u(t-m)\big)u(t-1)\end{aligned}$$

f and g are two separate networks. A feedback linearizing controller is obtained by calculating the controls according to:

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

Selecting the virtual control, $w(t)$, as an appropriate linear combination of past outputs plus the reference enables an arbitrary assignment of the closed-loop poles. As for the model-reference controller, feedback linearization is thus a nonlinear counterpart to pole placement with all zeros canceled (see Åström & Wittenmark, 1995). The principle is depicted in Figure 8.

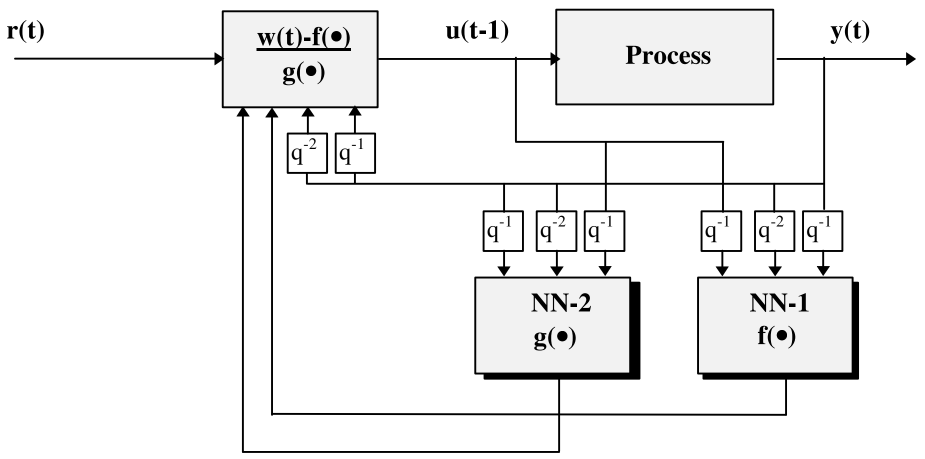


Figure 8 *Discrete feedback linearization.*

The design parameters must be specified in the file *fblinit*. Excerpts from the file are discussed below.

```
% ----------      Switches       -----------
regty      ='fbl';           % Controller type (fbl, pid, none)
```

`regty='fbl'` gives control by feedback linearization.

```
% ----- Neural Network Specification ------
% "nnfile" must contain the following variables which together define
% the network model:
% NN, NetDeff, NetDefg, W1f, W2f, W1g, W2g
% (i.e. regressor structure, architecture definitions, and weight matrices)
nnfile = 'net_file';         % Name of file
```

`nnfile`, which contains the regressor structure, architecture definition and weights of the neural network, must in this case include seven different variables due to the somewhat special network architecture. `NN` specifies the regressors and is set as for an NNARX model. `NetDeff` defines the architecture of the network implementing *f* while `NetDefg` defines the architecture of the network implementing *g*. `W1f` and `W2f` contain the weights for *f* while `W1g` and `W2g` contain the weights for *g*.

```
% ---- Desired characteristic polynomial ---
Am = [1 -1.4 0.49];          % Characteristic polynomial
```

A_m contains the desired closed-loop characteristic polynomial. In this case the polynomial has been selected to $A_m(z) = z^2 - 1.4z + 0.49$ corresponding to two poles in $z = 0.7$.

2.4 OPTIMAL CONTROL

Optimal Control	
opttrain	Simulate control by feedback linearization.
optrinit	File with design parameters for *opttrain.*
optcon	Simulate optimal control of process (similar to *invcon*).
optinit	File with design parameters for *optcon.*
opttest	Program for demonstrating the optimal control concept.

A simple training algorithm for design of optimal controllers has been implemented by a small modification of the specialized training algorithm implemented in *special2*. The modification consists of an additional term which is added to the criterion to penalize the 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 of inverse models and is also performed on-line. As for specialized training a “forward” model of the process must have been identified with the NNSYSID toolbox in advance.

The training is carried out with *opttrain* which implements a modified recursive Gauss-Newton algorithm. For the sake of brevity let

$$e_u(t) \equiv \frac{\partial y(t)}{\partial u(t-1)} e(t)$$

$$\psi_u(t) \equiv \frac{du(t-1)}{d\theta}$$

$$\psi(t) \equiv \frac{\partial \hat{y}(t)}{\partial u(t-1)} \psi_u(t)$$

The gradient is changed which manifests itself in a new weight update:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + P(t)\psi_u(t)\left(e_u(t) - \rho u(t-1)\right)$$

The update of the inverse Hessian is exactly the same as in the specialized training scheme implemented in *special2*. Naturally, it is an approximation to use the same covariance update(s) as in specialized training, but it appears to work reasonably well.

The following variables must be set in *optrinit* before the training is started:

```
% ----------      Switches       -----------
simul     = 'simulink';     % System specification (simulink/matlab/nnet)
method    = 'ff';           % Training algorithm (ff/ct/efra)
refty     = 'siggener';     % Reference generation (siggener/<variable name>)
```

`method` specifies the variation of the training algorithm (forgetting factor, constant trace, EFRA). `refty` specifies the type of reference signal. Since this is an on-line scheme related to specialized training, it is excellent for optimizing the controller for a prescribed reference trajectory.

```
% ----- Neural Network Specification ------
% The "forward model file" must contain the following variables which
% define a NNARX-model:
% NN, NetDeff, W1f, W2f
% and the "controller network file" must contain
% NetDefc, W1c, W2c
% (i.e. regressor structure, architecture definition, and weight matrices)
nnforw = 'forward';          % Name of file containing forward model
nnctrl = 'initopt';          % Name of file containing initial controller net
```

Two files must be present (actually the same file can be used for both networks): `nnforw` is set to the filename containing the network which models the process. This network must be trained in advance with the *nnarx* function in the NNSYSID toolbox. The variables defining the network must have the names `NN`, `NetDeff`, `W1f`, and `W2f`.

`nnctrl` contains the initial controller network. The network weights can for example be initialized with the generalized inverse training scheme (function *general*). The variables defining the network must have the names (`NN`,) `NetDefc`, `W1c`, and `W2c`.

If desired, it is possible to filter the reference:

```
% ------------ Reference filter ---------------
Am = [1 -0.7];               % Filter denominator
Bm = [0.3];                  % Filter numerator
```

`Am` and `Bm` do not specify a reference model as in specialized training. In this case they simply specify a filter to be used on the reference signal.

Most of the design parameters associated with the training algorithm were discussed in the section about specialized training. In this case it is of course necessary to include an additional parameter, namely ρ:

```
% ------------ Training parameters -----------
maxiter = 7;                 % Maximum number of epochs
rho     = 1e-3;              % Penalty factor on squared controls
```

If `rho=0`, the algorithm degenerates to specialized training and the inverse model is thus obtained.

When the controller network has been trained, the final network is stored in a file. *optcon* is then used for simulating the closed-loop system. This function is essentially identical to *invcon* except that the design parameters must be defined in *optinit* and the names of the variables specifying the controller network must have different names.

2.5 INSTANTANEOUS LINEARIZATION

Control by Instantaneous Linearization

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 solution by *diophant*.
apccon	Simulate approximate generalized predictive control.
apcinit	File with design parameters for *apccon.*
lintest	Program for demonstrating approximate pole placement control.

In Nørgaard et al. (2000) a technique for linearizing neural network models around the current operating point is given. The idea is summarized in the following.

Assume that a deterministic model of the process under consideration has been established with the NNSYSID-toolbox:

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

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

$$z(t) = \begin{bmatrix} y(t-1) & \cdots & y(t-n) & u(t-d) & \cdots & u(t-d-m) \end{bmatrix}^T$$

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

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

where

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

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

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

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

and

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

The approximate model may thus be interpreted as a linear model affected by an additional DC-disturbance, $\zeta(\tau)$, depending on the operating point.

It is straightforward to apply this principle to the design of control systems. The idea is illustrated in Figure 9.

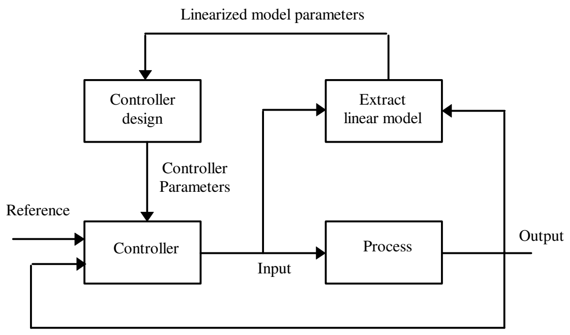


Figure 9. *Instantaneous linearization applied to control system design.*

Although the controller is not adaptive in the sense that it has been designed with time-varying processes in mind, the concept is closely related to the indirect self-tuning regulator concept examined in Åström & Wittenmark (1995). Instead of recursively *estimating* a linear model at each sample, a linear model is *extracted* from a nonlinear neural network model instead. The method is thus a type of gain scheduling control where the schedule is infinite. Unlike the previously mentioned concepts the controller is in this case not directly implemented by a neural network.

Approximate pole placement and minimum variance:

Together *lincon*, *lininit*, and *diophant* have adopted the instantaneous linearization principle for realization of different controllers. Three different controllers 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 prescribed 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 the criterion:

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

where I_t specifies the information gathered up to time t:

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

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

An excerpt of the file *lininit*:

```
% ----------      Switches       -----------
regty      ='rst';          % Controller type (rst/pid/none)
design     ='ppaz';         % Controller design (ppnz/ppaz/mv1/off)
```

The pole placement and minimum variance controllers are all realized in the RST-controller structure shown in fig. 10. Unless a constant gain PID-controller is desired, `regty` should thus be set to `'rst'`.

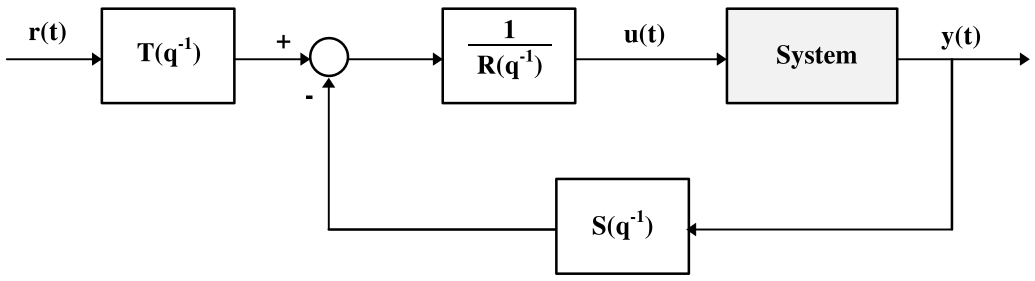


Figure 10 *The RST-controller:* $R(q^{-1})y(t) = T(q^{-1})r(t) - S(q^{-1})y(t)$

In Åström & Wittenmark (1995) this type of controller structure is discussed thoroughly and the reader is therefore refered to this book for supplementary information.

`design` specifies the type of design, i.e., the implementation of the "design block" in Figure 9:
`design='ppaz'` gives pole placement with all the zeros canceled.
`design='ppnz'` gives pole placement with no zeros canceled.
`design='mv1'` gives the MV1-controller.

If a pole placement design is selected, the following design parameters must be specified. The reader is referred to Åström & Wittenmark (1995) for an explanation of the different polynomials (for convenience the same notation is used here, except that Åström & Wittenmark uses the forward shift operator, q, rather than the delay operator, q^{-1}).

```
% -- Design parameters in pole placement --
% deg(Am)=deg(A)+deg(Ar)
% deg(Ao)=deg(A)-1         if no zeros are cancled
% deg(Ao)=deg(A)-deg(B)-1  if all zeros are canceled
Am = [1.0 -1.4 0.49 0];      % Denominator of desired model
Bm = [0.09];                 % Numerator of desired model (starts in z^{-1})
Ao = [1];                    % Observer polynomial
Ar = [1 -1];                 % Pre-specified factor of R. Ar MUST contain
                             % [1 -1] as a factor (=integrator).
As = 1;                      % Pre-specified factor of S (e.g., notch filter)
```

`Am`, `Bm`, `Ao`, `Ar`, and `As` are all polynomials in q^{-1}. `Bm` is only used if the zeros are canceled. If no zeros are canceled the open-loop zeros are retained and the content of `Bm` is ignored. However, the DC-gain is modified to ensure a unity gain at DC.

In the MV1 design it is only necessary to specify the penalty factor, δ:

```
% -------- Design parameters in MV1 --------
delta = 0.002;             % Penalty on squared differenced controls
```

The polynomials `Am`, `Bm`, `Ao`, `Ar`, and `As` are not used at all and do not have to be set.

Approximate GPC:

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

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

with respect to the N_u future controls

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

and subject to the constraint

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

N_1 denotes the minimum prediction (or costing) horizon, N_2 the maximum prediction (or costing) horizon, and N_u the (maximum) control horizon. ρ is a weighting factor for penalizing variations in the control input. $\zeta(\tau)$ is modelled as integrated white noise and the predictions of future outputs, $\hat{y}(t+i)$, are determined as the minimum variance predictions. The optimization problem (which must be solved on-line since a new linear model is obtained 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. Nørgaard et al. (2000) details the derivation of the controller.

The design parameters for the approximate GPC are relatively straightforward to initialize. The following is an excerpt from the file *apcinit*. Compare this to the definition of the GPC-criterion J_5.

```
% ----------      Switches       -----------
regty      ='apc';           % Controller type (apc, pid, none)

% ---------- APC initializations -----------
N1 = 1;                      % Minimum prediction horizon (typically=nk)
N2 = 7;                      % Maximum prediction horizon (>= nb)
Nu = 2;                      % Control horizon
rho = 0.03;                  % Penalty on squared differenced controls
```

2.6 NONLINEAR PREDICTIVE CONTROL

Nonlinear Predictive Control	
npccon1	Simulate NPC using a Quasi-Newton method.
npccon2	Simulate NPC using a Newon based Levenberg-Marquardt method.
npcinit	File containing the design parameters.
predtest	Program for demonstrating generalized predictive control.

The instantaneous linearization technique has some shortcomings when the nonlinearities are not relatively smooth. Unfortunately, practically relevant criterion-based design methods founded directly on nonlinear neural network models are few. One of the most promising methods is nonlinear predictive control, which is based on the criterion J_5 defined above. However, in nonlinear predictive control the prediction of future outputs is not obtained through a linearization, but from succesive recursive use of a nonlinear NNARX model:

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

The optimization problem is in this case much more difficult to solve and an iterative search method is required. The reader is referred to Nørgaard et al. (2000) for a derivation of the control law and for a discussion of relevant optimization algorithms. The toolkit offers two different algorithms for solving the problem: A Quasi-Newton method applying the BFGS-algorithm for updating the inverse Hessian and a Newton based Levenberg-Marquardt method.

An excerpt from the file *npcinit.m* shows that most design parameters are similar to *apcinit.m*:

```
% ----------      Switches       -----------
regty      ='npc';    % Controller type (npc, pid, none)

% ---------- GPC initializations -----------
N1 = 1;               % Min. prediction horizon (Must equal the time delay)
N2 = 7;               % Max. prediction horizon (>= nb)
Nu = 2                % Control horizon
rho = 0.03;           % Penalty on squared differenced controls

% -- Minimization algorithm initializations -
maxiter = 5;                 % Max. number of iterations to determine u
delta = 1e-4;                % Norm-of-control-change-stop criterion
initval = '[upmin(Nu)]';
```

Different design parameters specify how the optimization should be performed. The default values shown in the excerpt will work for most applications, but sometimes it can be necessary to modify them.

`maxiter` is of course the maximum number of iterations. `maxiter=5` is usually appropriate. A smaller value implies that the simulation runs faster while a bigger value implies an enhanced accuracy.

`delta` is essentially another stopping criterion. If the length of the vector obtained by subtracting the previous iterate from the current is smaller than `delta`: $\left|U^{(i)}(t)-U^{(i-1)}(t)\right| < delta$ the current iterate is accepted as the final one.

`initval` is a vector that has been introduced because it may happen that the criterion to be minimized (J_5) has more than one local minimum. Since the criterion is minimized iteratively an initial guess on the sequence of future controls, $U^{(0)}(t)=\left[u^{(0)}(t),u^{(0)}(t+1),\ldots,u^{(0)}(t+N_u-1)\right]$, is required. If the criterion has more than one local minimum it is desireable to execute the optimization algorithm more than once to find the global one, starting from different initializations of $U^{(0)}(t)$. *npccon1* and *npccon2* have been implemented in such a way that the first N_u-1 controls are taken from the final iterate on $U(t$-1), determined in the previous sample. The vector `initval` then contains the initial guess(es) on $u(t+N_u$-1). `initval` has been implemented as a "string vector" to make it possible that expressions can be written in the vector. The default choice is `initval = '[upmin(Nu)]'` which implies that $U^{(0)}(t)=\left[u_{t-1}^{(final)}(t),u_{t-1}^{(final)}(t+1),\ldots,u_{t-1}^{(final)}(t+N_u-2),u_{t-1}^{(final)}(t+N_u-2)\right]$ is used (the final control in the vector $U^{(\text{final})}(t$-1) is repeated). The choice `initval='[upmin(Nu) 5]'` implies that the algorithm is executed twice and that the starting point the second time is

$$U^{(0)}(t)=\left[u_{t-1}^{(final)}(t),u_{t-1}^{(final)}(t+1),\ldots,u_{t-1}^{(final)}(t+N_u-2),5\right]$$

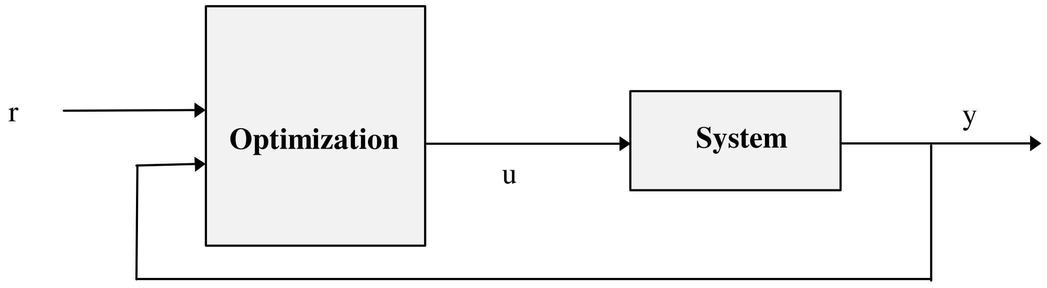


Figure 11 *Nonlinear predictive control. The controller is an iterative optimization scheme.*

References

R. Fletcher (1987): *“Practical Methods of Optimization,”* Wiley, 1987

The MathWorks (1999): *“Control System Toolbox, User’s Guide, Version 4.2”* The MathWorks, Inc.

K.J. Hunt, D. Sbarbaro (1991): *“Neural Networks for Nonlinear Internal Model Control,”* IEE Proceedings-D, Vol. 138, No. 5, pp. 431-438.

K.J. Hunt, D. Sbarbaro, R. Zbikowski, P.J. Gawthrop (1992): *“Neural Networks for Control Systems - A Survey,”* Automatica, Vol. 28, No. 6, pp. 1083-1112.

L. Ljung (1987):*“System Identification - Theory for the User,”* Prentice-Hall, 1987.

M. Nørgaard (2000): *“Neural Network Based System Identification Toolbox, Ver. 2”* Tech. report 00-E-891, Department of Automation, Technical University of Denmark.

M. Nørgaard, O. Ravn, N. K. Poulsen, L. K. Hansen (2000): *“Neural Networks for Modelling and Control of Dynamic Systems,”* Springer-Verlag, London, 2000.

D. Psaltis, A. Sideris, A.A. Yamamure (1988): *“A Multilayered Neural Network Controller,”* Control Sys. Mag., Vol. 8, No. 2, pp. 17-21.

M.E. Salgado, G. Goodwin, R.H. Middleton (1988): *“Modified Least Squares Algorithm Incorporating Exponential Forgetting and Resetting,”* Int. J. Control, 47, pp. 477-491.

K.J. Åström, B. Wittenmark (1995): *“Adaptive Control,”* 2nd. Edition, Addison-Wesley.