The Dominant Pole Design Toolbox

Persson, Per

1992

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

ISSN 0280–5316
ISRN LUTFD2/TFRT--7497--SE

The Dominant Pole Design Toolbox

Per Persson

Department of Automatic Control
Lund Institute of Technology
December 1992

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name

Date of issue
December 1992

Document Number
ISRN LUTFD2/TFRT--7497--SE

Author(s)
Per Persson

Supervisor

Sponsoring organisation

Title and subtitle
The Dominant Pole Design Toolbox

Abstract

This report describes the Matlab routines that were used in the thesis Persson, P.: *"Towards Autonomous PID Control."* The routines are used to compute PID controllers based on placement of a few dominant poles. The user must specify a transfer function of the plant to be controlled and the desired degree of stability of the closed loop system. The stability is expressed as the maximum of the sensitivity function of the plant. From these specifications are the PID controller parameters computed, by minimizing the cost functional $\mathrm{IE} = \int_0^\infty e(t)\,dt$.

The routines can also be used for computing frequency responses, amplitude and phase margins, sensitivity functions, etc.

A method of connecting Matlab and Simnon is also briefly described. By this connection, Simnon can be used as a computation engine for Matlab. Simnon commands can be given from Matlab, parameters can be set, values in Simnon can be retrieved to Matlab, and simulation results can be taken from Simnon into Matlab.

Key words
PID control, Controller Design, Controller Tuning

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
28

Recipient's notes

Security classification

1. Introduction

Ever since the publication of the famous articles [Ziegler and Nichols, 1942] and [Ziegler and Nichols, 1943] there has been an interest in systematic tuning methods for PID controllers. This interest has increased recently due to research on automatic tuning of controllers. See, e.g., the article [Åström and Hägglund, 1984] and the book [Åström and Hägglund, 1988]. Several controllers with automatic tuning are commercially available.

The PhD thesis [Persson, 1992b] describes methods for the tuning of PID controllers based on pole placement. During the work with the thesis a number of routines for computation of PID controllers were written. This report presents an improved version of these routines as a Matlab toolbox. For readings on Matlab, see [MathWorks, 1990]. The toolbox will be called the DPD Toolbox (the Dominant Pole Design Toolbox). This report is intended to be a manual for the DPD Toolbox, but also to further illustrate the computations carried out in [Persson, 1992b].

The methods require that the transfer function of the process is known. The toolbox contains design routines and also some utility functions of general interest. Apart from using the routines in the toolbox for computing controllers there are also routines for computing frequency responses, amplitude and phase margins, sensitivity functions, etc.

This report also describes a way to connect Matlab and Simnon. By doing this Simnon can be used as a computing engine for Matlab. There are routines for giving Simnon commands from Matlab, to set Simnon parameters from matlab, to retrieve Simnon values to Matlab, and to get time signals from Simnon to Matlab.

2. Design Methods

This chapter will give a brief review of the results and methods presented in the thesis [Persson, 1992b]. We also give references to the implementation of the different methods.

Design of PID controllers

All methods that are presented here are based on the placement of a few (two or three) of the dominant poles of the closed loop system. We will use the parallel form of the PID controller

$$U(s) = k(\beta Y_r(s) - Y(s) + \frac{1}{sT_i}(Y_r(s) - Y(s)) - \frac{sT_d}{1 + s\frac{T_d}{N}}Y(s)), \qquad (2.1)$$

where $u(t)$ is the controller output, $y(t)$ is the plant output, $y_r(t)$ is the reference signal, β is the set point weighting factor, N is a filter constant, and k, T_i, T_d are the controller parameters. The parameter β will affect a zero of the closed loop system, but does not affect the stability of the system. Notice that the derivative only acts on the output from the process, $y(t)$. The parameters β and N are not determined by pole placement. For computation of the poles of the system the transfer function

$$G_c(s) = k(1 + \frac{1}{sT_i} + sT_d) = k + \frac{k_i}{s} + sk_d \qquad (2.2)$$

will be used. The transfer function of the plant will be denoted $G_p(s)$, and is assumed to be known.

We will require that the characteristic equation, $1 + G_c(s)G_p(s) = 0$, has roots in specified locations. For controllers with two parameters we specify poles in the locations $p_{1,2} = \omega_0(-\zeta_0 \pm i\sqrt{1-\zeta_0^2})$ and for controllers with three parameters we specify poles in the locations $p_{1,2} = \omega_0(-\zeta_0 \pm i\sqrt{1-\zeta_0^2})$ and $p_3 = -\alpha_0\omega_0$. The equations $1 + G_c(p_i)G_p(p_i) = 0$ are then solved with respect to the controller parameters. The controller parameters will be functions of ω_0, ζ_0, and α_0. The design routines will accept $\zeta_0 > 1$ which corresponds to two poles on the negative real axis.

The control error is defined as $e(t) = y_r(t) - y(t)$. It is easy to show that

$$\text{IE} = \int_0^\infty e(\tau)\,d\tau = \frac{1}{k_i}, \qquad (2.3)$$

when the input to the closed system is a step on the input of the plant. The integrated error, IE, can be used as an optimization criterion. This is advantageous because we have a simple analytical expression of the cost functional. Since the distance of the poles from the origin is closely coupled to the performance of the system we have chosen to determine ω_0 by maximizing k_i.

For a well damped closed system IE $\approx$ IAE $= \int_0^\infty |e(\tau)|\,d\tau$, which is the conventional optimization criterion.

To determine a suitable value of ζ_0 we will specify a value of the maximum of the sensitivity function M_s, defined as

$$M_s = \max_{\omega>0} \frac{1}{|1 + G_c(i\omega)G_p(i\omega)|}.$$

This way of determining ζ_0 gives good control of the shape of the closed loop step responses. A small value of M_s will give a well damped system and a large value will give an oscillatory system. Normally a value in the interval 1.5...2.0 is recommended. It is not recommended to choose a value of ζ_0 directly, since the result of design with ζ_0 directly will vary much depending on the process.

The parameter α_0 must be chosen for the design of PID controllers. Normally $\alpha_0 = 1$ is a good choice. For resonant processes it is necessary to choose α_0 smaller, $\alpha_0 \approx 0.2 \ldots 0.5$.

Design of PI controllers In case of a PI controller the formulas for k and k_i become

$$k = -\frac{\sqrt{1-\zeta_0^2}A + \zeta_0 B}{\sqrt{1-\zeta_0^2}(A^2+B^2)} \tag{2.4}$$

$$k_i = -\frac{\omega_0 B}{\sqrt{1-\zeta_0^2}(A^2+B^2)}, \tag{2.5}$$

where $A = \operatorname{Re} G_p(\omega_0(-\zeta_0 + i\sqrt{1-\zeta_0^2}))$ and $B = \operatorname{Im} G_p(\omega_0(-\zeta_0 + i\sqrt{1-\zeta_0^2}))$. The expressions for $\zeta_0 > 1$ look similar, but require that $G_p(s)$ is evaluated twice. The functions $k(\omega_0, \zeta_0)$ and $k_i(\omega_0, \zeta_0)$ are implemented in the function 'dppi'. The recommended design method is to choose a ζ_0, then maximize $k_i(\omega_0, \zeta_0)$ with respect to ω_0, giving $k^*(\zeta_0)$ and $k_i^*(\zeta_0)$. The parameter ζ_0 should be chosen such that the system gets a prescribed M_s-value. This means solving a nonlinear equation with respect to ζ_0. The equation becomes

$$M_s = \max_{\omega>0} \frac{1}{|1 + (k^*(\zeta_0) + \frac{k_i^*(\zeta_0)}{i\omega})G_p(i\omega)|}. \tag{2.6}$$

This is an approximate, but often sufficiently accurate computation method. This is implemented by the function '**pidesms**'.

Another possibility is to determine a function $\zeta_0 = \zeta_0(\omega_0)$ from the equation

$$M_s = \max_{\omega>0} \frac{1}{|1 + (k(\omega_0, \zeta_0(\omega_0)) + \frac{k_i(\omega_0, \zeta_0(\omega_0))}{i\omega})G_p(i\omega)|} \tag{2.7}$$

and then maximize $k_i = k_i(\omega_0, \zeta_0(\omega_0))$ with respect to ω_0. This is implemented by the function '**pides2**'. This is computationally ***very*** demanding, but gives the controller with maximal k_i for a specified M_s. The approximate method usually gives results very close to the true optimum. For systems with poorly damped poles it is necessary to use the exact design method.

Routines for design of PI controllers with ζ_0 chosen to get specified values of A_m and φ_m are also available, '**pidesam**' and '**pidespm**'. Their implementation is very similar to the one of '**pidesms**', ζ_0 is chosen to get the prescribed value of A_m or φ_m.

Design of PD controllers In case of a PD controller the formulas for k and k_d become

$$k = \frac{-\sqrt{1-\zeta_0^2}A + \zeta_0 B}{\sqrt{1-\zeta_0^2}(A^2+B^2)} \tag{2.8}$$

$$k_d = \frac{B}{\omega_0\sqrt{1-\zeta_0^2}(A^2+B^2)}, \tag{2.9}$$

where $A = \mathrm{Re}\, G_p(\omega_0(-\zeta_0 + i\sqrt{1-\zeta_0^2}))$ and $B = \mathrm{Im}\, G_p(\omega_0(-\zeta_0 + i\sqrt{1-\zeta_0^2}))$. The expressions for $\zeta_0 > 1$ look similar, but require that $G_p(s)$ is evaluated twice. The functions $k(\omega_0, \zeta_0)$ and $k_d(\omega_0, \zeta_0)$ are implemented in the function 'dppd'.

In the PD case it is natural to maximize the controller constant k, to get minimum offset to a load disturbance. Design with specified value of M_s is implemented in the function 'pddesms' in the same way as in 'pidesms'.

Design of PID controllers In the case of a PID controller the formulas for k, k_i, and k_d become

$$k = -\frac{\sqrt{1-\zeta_0^2}(-2\alpha_0\zeta_0(A^2+B^2) + (1+\alpha_0^2)AC) + (\alpha_0^2-1)\zeta_0 BC}{(1-2\alpha_0\zeta_0+\alpha_0^2)\sqrt{1-\zeta_0^2}(A^2+B^2)C} \tag{2.10}$$

$$k_i = -\alpha_0\omega_0\frac{(\alpha_0-\zeta_0)BC + \sqrt{1-\zeta_0^2}(AC - A^2 - B^2)}{(1-2\alpha_0\zeta_0+\alpha_0^2)\sqrt{1-\zeta_0^2}(A^2+B^2)C} \tag{2.11}$$

$$k_d = -\frac{(\alpha_0\zeta_0 - 1)BC + \alpha_0\sqrt{1-\zeta_0^2}(AC - A^2 - B^2)}{\omega_0(1-2\alpha_0\zeta_0+\alpha_0^2)\sqrt{1-\zeta_0^2}(A^2+B^2)C}, \tag{2.12}$$

where $A = \mathrm{Re}\, G_p(\omega_0(-\zeta_0 + i\sqrt{1-\zeta_0^2}))$, $B = \mathrm{Im}\, G_p(\omega_0(-\zeta_0 + i\sqrt{1-\zeta_0^2}))$, and $C = G_p(-\alpha\omega_0)$. The expressions for $\zeta_0 > 1$ look similar, but require that $G_p(s)$ is evaluated three times. The functions $k(\omega_0, \zeta_0, \alpha_0)$, $k_i(\omega_0, \zeta_0, \alpha_0)$, and $k_d(\omega_0, \zeta_0, \alpha_0)$ are implemented in the function 'dppid'.

The recommended design method is very similar to the one for PI controllers. The only difference is that we must also have a term $i\omega k_d^*(\zeta_0)$ in the equation for M_s. The recommended design method is implemented in the function 'piddesms'.

A second method for PID controller design is described in [Persson, 1992b] on pages 97 – 101. The method consists of modifying a well tuned PI controller by increasing the derivative gain k_d, until the M_s-function reaches a specified value. This design method is implemented in the function 'piddesm2'. In general this method is inferior to the one implemented in 'piddesms'.

3. Implementation

In this chapter the Matlab implementation of the design routines will be discussed. The data structures will be described, and the structure of the design and utility functions will be presented.

3.1 Hardware and Software

The implementation which is described here works under Matlab 4.0 on a SPARC. Versions of the routines for Matlab 3.5j on SPARC, PC 386, and PC 486 are also available, but are not described here. All function names have been limited to eight characters to make it possible to use the functions unchanged on PC systems.

3.2 Data Structures

The only data structure available in Matlab is the matrix. Everything in Matlab must be expressed with matrices. In this section the conventions used in the DPD Toolbox will be described.

Transfer functions Transfer functions are described as strings, where the 's' is the Laplace transform variable. All Matlab math operators and functions are allowed in the transfer functions strings.

The design routines require that these strings can be evaluated when 's' assumes the value of a vector, hence the multiplicative operators '*', '^', '/', and '\' must be replaced by the operators '.*', '.^', './', and '.\'. The function 'convert' does this transformation.

EXAMPLE 3.1
The transfer function

$$G(s) = \frac{e^{-s}}{(s+1)^2} \tag{3.1}$$

can be described as a string as

```
pstr = 'exp(-s)./(s+1)./(s+1)'
```

or

```
pstr = 'exp(-s)./(s+1).^2'
```

or

```
pstr = 'exp(-s).*(s+1).^(-2)'.
```

□

EXAMPLE 3.2
The transfer function

$$G(s) = \frac{1}{(s+1)^8} \tag{3.2}$$

is described as

```
pstr = '(s+1).^(-8)' or pstr = '1./(s+1).^8'.
```

□

In Matlab Version 3, it is possible to use global variables in the definition of transfer function strings. This is not possible in Matlab Version 4, without changes in the code. To use global variables in the transfer function strings, the variables should be declared '`global`' in the routine '`evals`'.

The Controller Data Structure The design routines return a row vector describing the controller. The vector has the content

$$c = [\omega_0 \quad \zeta_0 \quad \alpha_0 \quad k \quad k_i \quad T_i \quad k_d \quad T_d \quad N_r]. \tag{3.3}$$

The parameter ω_0 is the distance of the dominant poles from the origin, ζ_0 is their relative damping. In case that the design routine uses a third pole $-\alpha_0\omega_0$, the value of α_0 is returned, in other cases `NaN` is returned. The controller parameters are returned as k, k_i, T_i, k_d, and T_d. The parameter N_r tells which of the fundamental pole placement routines has computed the controller. This parameter is never used. This data structure should be accessed by the access functions '`getw0`', '`getz0`', '`geta0`', '`getk`', '`getki`', '`getti`', '`getkd`', and '`gettd`'. The routine '`con2str`' converts the controller data structure to the controller transfer function expressed as a string.

Frequency responses Frequency responses generated with '`sfrcol`' are stored as

$$\begin{bmatrix} \omega_1 & G_1(\omega_1) & \dots & G_m(\omega_1) \\ \vdots & \vdots & & \vdots \\ \omega_n & G_1(\omega_n) & \dots & G_m(\omega_n) \end{bmatrix}. \tag{3.4}$$

This data structure is also used in the toolboxes described in [Gustafsson *et al.*, 1990a] and [Gustafsson *et al.*, 1990b], which are recommended for plotting.

3.3 Utility Routines

The transfer functions represented as strings are evaluated with the function '`evals`',

EXAMPLE 3.3—The use of `evals`
To evaluate the transfer function $G(s) = e^{-s}/(s+1)^2$, use the following commands.

```
>> pstr = 'exp(-s)./(s+1)./(s+1)';
>> w = [0.1 0.2 0.3];
>> evals(pstr, i*w)

ans =

   0.9461 - 0.2920i   0.7964 - 0.5388i   0.5825 - 0.7088i
```

□

There are routines for computing the maximum of the sensitivity function, '`mscl`', amplitude margin, '`amarg`', and phase margin, '`pmarg`'. The routines for computing amplitude and gain margins are based on the general routines '`asolveol`' and '`psolveol`' which solve the equations

$$a_1 = |G_p(i\omega)G_c(i\omega)| \tag{3.5}$$

$$a_2 = \arg G_p(i\omega)G_c(i\omega) \tag{3.6}$$

with respect to ω for given a_1 and a_2.

All these routines require the process and the controller expressed as strings. There is a conversion routine that converts the controller data structure to a string, 'con2str'.

The function 'makep' returns one of a number of standard processes with parameters given as arguments to the function. The following processes are provided

$$G_1 = \frac{p_1 e^{-sp_2}}{sp_3 + 1} \tag{3.7}$$

$$G_2 = \frac{1}{(s+1)^{p_1}} \tag{3.8}$$

$$G_3 = \frac{(1 - sp_1)}{(s+1)^3} \tag{3.9}$$

$$G_4 = \frac{e^{-sp_1}}{(sp_2 + 1)(sp_3 + 1)} \tag{3.10}$$

$$G_5 = \frac{p_2^2 e^{-sp_1}}{(s^2 + 2sp_2p_3 + p_2^2)} \tag{3.11}$$

$$G_6 = \frac{p_2^3 p_4 e^{-sp_1}}{(s + p_2p_4)(s^2 + 2sp_2p_3 + p_2^2)} \tag{3.12}$$

$$G_7 = \frac{1}{(s+1)(sp_1 + 1)(sp_1^2 + 1)} \tag{3.13}$$

$$G_8 = \frac{1}{(s+1)(sp_1 + 1)(sp_1^2 + 1)(sp_1^3 + 1)} \tag{3.14}$$

$$G_9 = \frac{(1 - sp_1/2)}{(1 + sp_1/2)(1 + sp_2)} \tag{3.15}$$

$$G_{10} = \frac{e^{-sp_1}}{(1 + sp_2)(1 + sp_3)(1 + sp_4)} \tag{3.16}$$

$$G_{11} = \frac{e^{-sp_1}}{s(sp_2 + 1)} \tag{3.17}$$

$$G_{12} = \frac{e^{-sp_1}}{s(sp_2 + 1)(sp_3 + 1)}. \tag{3.18}$$

Many of the routines require a tolerance for the solution of an equation or an optimization. When a tolerance is required in a function it can be given as a parameter to the function. If the tolerance parameter is omitted the tolerance is taken from the global variable 'GTOL', and if 'GTOL' is not defined the tolerance is given from the function 'deftol'. Normally the tolerance is given from 'deftol' and is set to 10^{-4} as default.

The optimization is carried out with the functions 'opt' and 'optg'. These functions implement a golden ratio optimization algorithm.

The solution of an equation is carried out by the functions 'solve' and 'solveb'. In these functions a simple bisection method is implemented. The function 'solveb' is specially implemented for solving phase equations to get the correct phase value across $n\pi$ borders.

3.4 Design Routines

The design routines are implemented in a number of layers, see Figure 3.1.

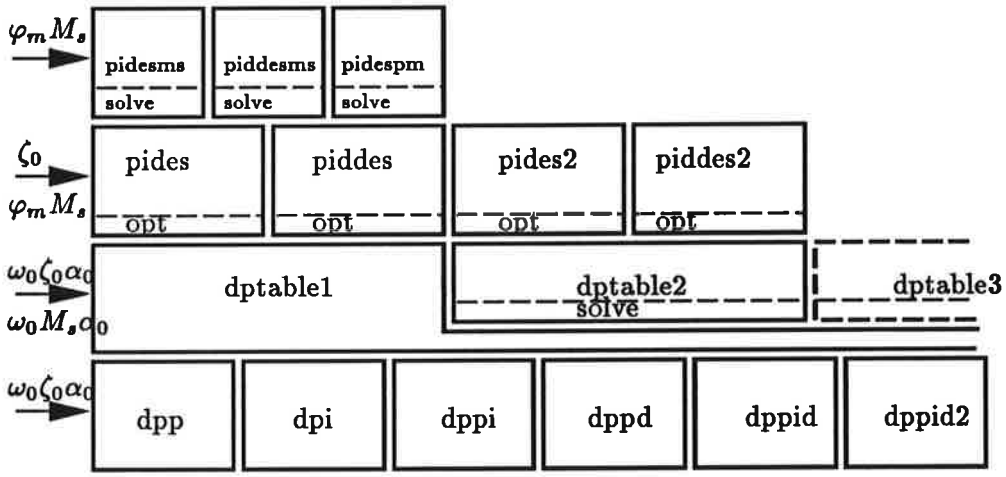

Figure 3.1 The structure of the design routines.

The lowest level consists of the basic pole placement routines, i.e., routines that compute controller parameters given ω_0, ζ_0, and α_0. The next level consists of interface routines to the basic primitives. There may be several of these interface routines. In this version four interface routines are available, '`dptable1`', '`dptable2`', '`dptable3`', and '`dptable4`'. In '`dptable1`' controllers are computed given ω_0, ζ_0, and α_0, ω_0 may be a vector. In '`dptable2`' controllers are computed given ω_0, M_s, and α_0. This involves solving the M_s-equation with respect to ζ_0. The routine '`dptable3`' takes the input parameters ω_0, A_m, α_0 and '`dptable4`' takes the input parameters ω_0, φ_m, α_0

The third level optimizes k_i, given a ζ_0 or M_s. Finally, the fourth level solves equations with respect to ζ_0 such that some stability requirements are fulfilled. These requirements may be M_s, A_m, or φ_m. These are the approximate, but recommended design functions.

4. Examples

This chapter will present a number of examples of the use of the design routines, and some discussions of the examples. All examples in this chapter have been executed in Matlab Version 4.0 on an SPARC ELC, exactly as they are written here.

4.1 Hints for the use of the routines

When design is carried out with M_s as design parameter the default search interval of ζ_0 is '`linspace(0.01, 2, 10)`', this can handle most cases, but requires a lot of computation. If the plant is well damped and we require a moderate M_s 1.6 to 2, say, then a smaller search interval can be supplied, e.g., $\zeta_0 = [0.1\ 0.5\ 0.9]$

The ω_0 should be chosen such that we are sure to detect a maximum. The computations are relatively cheap, don't use a too sparse discretization.

Normally a tolerance of 10^{-4} in the computations is sufficient. A larger tolerance may cause errors in the routines.

Problems may also appear if the process transfer function is evaluated at a process pole. To avoid this choose the initial guesses of ω_0 ans α_0 'irregularly.' For example, if the routine '`piddesms`' is used for $G_p(s) = e^{-s}/(s+1)$ choose `w0s = [0.11:0.1:3.0]` rather than `w0s = [0.1:0.1:3.0]` or `alpha = 1.01` rather than `alpha = 1.0`.

Observe that the optimization is meaningless for processes of too low order. For a second order process we can place the poles anywhere with a PID controller, and get the closed system arbitrarily fast.

In Example 4.1 the function '`polyadd`' is used. The function is defined as:

```
function p = polyadd(p1, p2)
dn = length(p1) - length(p2);
p  = [zeros([1 -dn]) p1] + [zeros([1 dn]) p2];
```

4.2 Examples

In the following a number of examples of the use of the routines will be presented. The reader may need to consult Chapter 5 for a brief description of the function.

EXAMPLE 4.1—Controllers with different ζ_0
In the following example a number of PI controllers are designed with different values of $\zeta_0$for the process

$$G(s) = \frac{1}{(s+1)^4}.$$

The corresponding family of step responses and control signals are simulated and plotted.

```
pstr = '1./(s+1).^4';
b = 1; a = conv([1 2 1], [1 2 1]);
tv = [0:0.1:40]; w0s = [0.1:0.1:2]; z0s = [0.1:0.1:0.9];
for ix = z0s,
  con = pides(pstr, w0s, ix);
  r = [1 0]; s = [getk(con) getki(con)]; t = s;
  ysp = step(conv(b, t), polyadd(conv(a, r), conv(b, s)), tv);
  usp = step(conv(a, t), polyadd(conv(a, r), conv(b, s)), tv);
  subplot(2, 1, 1); hold on; plot(tv, ysp); drawnow;
  subplot(2, 1, 2); hold on; plot(tv, usp); drawnow;
end;
subplot(2, 1, 1);
grid; xlabel('Time'); ylabel('Output signal');
subplot(2, 1, 2);
grid; xlabel('Time'); ylabel('Control signal');
```

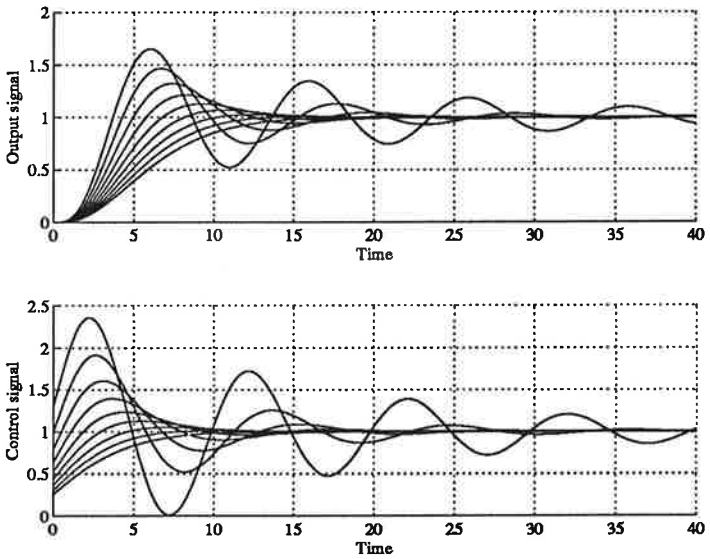

Figure 4.1 Step responses of $G(s) = 1/(s+1)^4$ controlled by PI controllers designed with different ζ_0.

Designing controllers with ζ_0 as the design parameter may produce very different looking step responses with the same ζ_0 for different processes. For this reason it is better to design controllers with specifications on M_s.

In this case the simulation is carried out by computing the transfer functions and using the standard function `'step'`. □

EXAMPLE 4.2—Computation of A_m and φ_m
In this example we will design PI controllers for

$$G(s) = \frac{e^{-sL}}{s+1}$$

with different values of the design parameter M_s. Phase and amplitude margins are then computed for the final systems as functions of M_s.

```
dels = [0.1 0.5 1];
mss = [1.8:0.2:3.0]; ws = [0.1:0.1:20]; res = [];
for jx = dels,
  tmp = [];
  pstr = makep(1, 1, jx, 1);
  for ix = mss,
    con = pidesms(pstr, ws, ix);
    tmp = [tmp;
           amarg(pstr, con2str(con), ws) ...
           pmarg(pstr, con2str(con), ws)];
  end;
  res = [res tmp];
end;
c1 = 1:2:cols(res); c2 = 2:2:cols(res);
subplot(2, 1, 1); plot(mss, res(:, c1));
xlabel('Ms'); ylabel('Am');
text(mss(2), res(2, c1(1)), 'L=0.1')
text(mss(2), res(2, c1(2)), 'L=0.5')
text(mss(2), res(2, c1(3)), 'L=1.0')
subplot(2, 1, 2); plot(mss, res(:, c2));
xlabel('Ms'); ylabel('Pm');
text(mss(2), res(2, c2(1)), 'L=0.1')
text(mss(2), res(2, c2(2)), 'L=0.5')
text(mss(2), res(2, c2(3)), 'L=1.0')
```

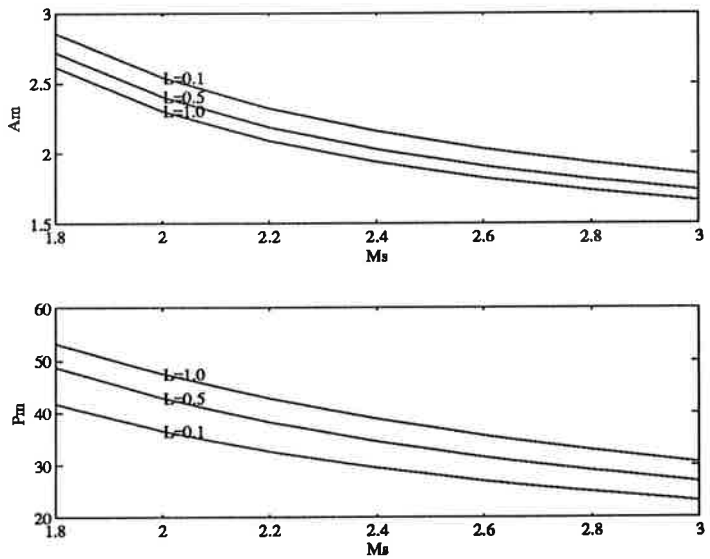

Figure 4.2 Amplitude and phase margins for systems designed with different M_s.

This example illustrates that a constant A_m or φ_m as design parameter may give very different M_s in the closed system for different processes. □

EXAMPLE 4.3—A process with different parameters
In this example the process

$$G_p(s) = \frac{1 - \alpha s}{(s+1)^3}$$

is controlled by a PI controller designed with $M_s = 2$ for different values of the parameter α. Load responses and the corresponding control signals are simulated and plotted.

```
clg;
tv = [0:0.1:30]; alphas = [0:0.5:2]; w0s = [0.1:0.1:2];
for ix = alphas,
  pstr = makep(3, ix);
  b = [-ix 1];
  a = [1 3 3 1];
  con = pidesms(pstr, w0s, 2);
  r = [1 0]; s = [getk(con) getki(con)]; t = s;
  ysp = step(conv(b, r), polyadd(conv(a, r), conv(b, s)), tv);
  usp = step(-conv(b, s), polyadd(conv(a, r), conv(b, s)), tv);
  subplot(2, 1, 1); hold on; plot(tv, ysp); drawnow;
  subplot(2, 1, 2); hold on; plot(tv, usp); drawnow;
end;
subplot(2, 1, 1); grid; xlabel('Time');
ylabel('Output signal');
subplot(2, 1, 2); grid; xlabel('Time');
ylabel('Control signal');
```

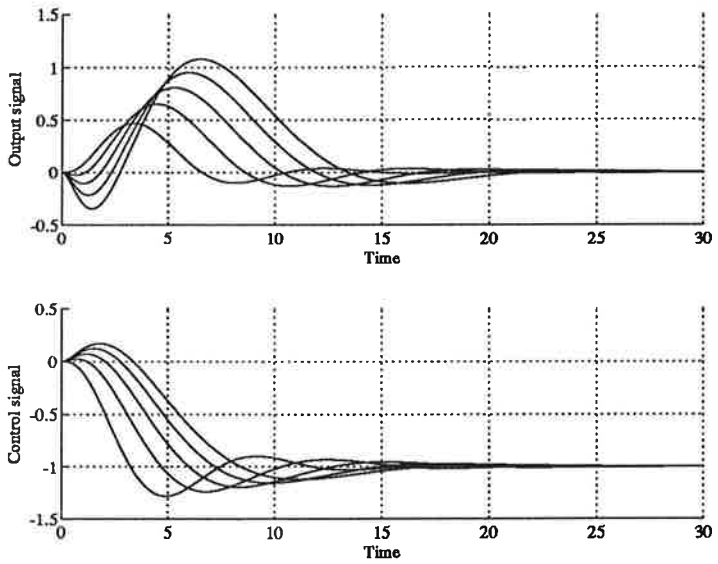

Figure 4.3 Load responses for $G_p(s) = (1 - s\alpha)/(s+1)^3$ for different values of α with PI controllers designed with $M_s = 2$.

The non-minimum phase character of the responses can clearly be seen for large values of α. □

EXAMPLE 4.4—Nyquist curves from PI design on different processes
In this example we design PI controllers for a number of different processes. The processes are described in the array 'a', which contains the process number and the parameters of the process.

PI controllers are designed with constant ζ_0 and with constant M_s. As can be seen from the plotted Nyquist curves we get much more similarity of different loop transfer functions with controllers designed with M_s than with constant ζ_0.

```
w = logspace(-1, 2, 200)'; resms = [w]; resz = [w];
a = [1 1 1 1
     2 4 0 0
     3 1 0 0
     4 1 2 2];
for ix = a'
  pstr = makep(ix');
  cms = pidesms(pstr, [0.1:0.1:10], 2);
  cz  = pides(pstr, [0.1:0.1:10], 0.5);
  f = sfrcol(pstr, con2str(cms), w);
  resms = [resms f(:, 2)];
  f = sfrcol(pstr, con2str(cz), w);
  resz = [resz f(:, 2)];
end;
nypl(resms); nygrid;
plotc(-1, 1/2, -pi/2, pi/2);
axis([-1 0.5 -1 0.5])
nypl(resz); nygrid;
plotc(-1, 1/2, -pi/2, pi/2);
axis([-1 0.5 -1 0.5])
```

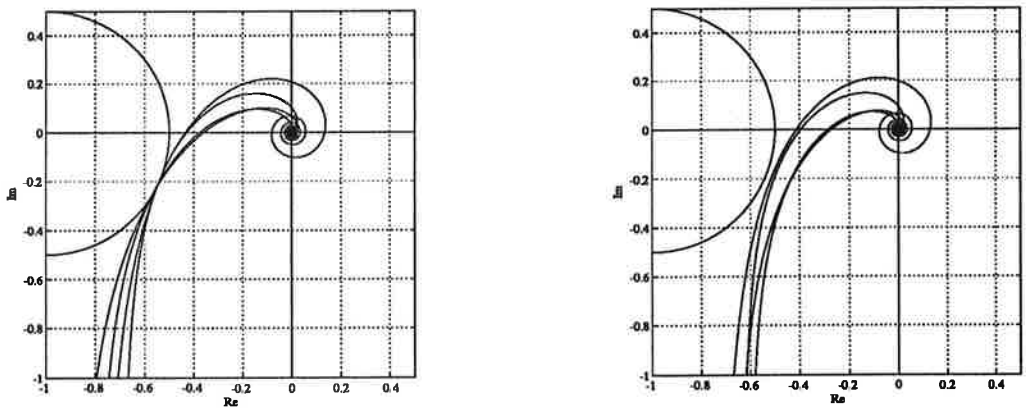

Figure 4.4 To the left: Nyquist plots for processes designed with constant M_s, to the right: Nyquist plots for processes designed with constant ζ_0.

EXAMPLE 4.5—Timing
In this example controllers are computed with different tolerance in the optimization and equation solving routines, and the time it takes to compute the different controllers are recorded.

```
pstr = 'exp(-s)./(s + 1)'; w0s = [0.1:0.1:10]; res = [];
for ix = 2:8
  tic;
  c = pidesms(pstr, w0s, 2, [], 10^(-ix));
  t1 = toc;
  tic;
  c = pidesms(pstr, w0s, 2, [0.2:0.2:0.6], 10^(-ix));
  t2 = toc;
  res = [res; t1 t2];
end;
res

res =

   11.0203    9.1207
   18.1734   15.8076
   28.4095   25.4336
   39.2039   35.6356
   52.6931   48.4410
   69.8643   65.0888
   93.5680   83.4929
```

Some time can be saved by guessing a correct interval for ζ_0 instead of relying on the default values from the algorithm. □

EXAMPLE 4.6—Computation of $k_i(k)$ curves
This is an example where we need to use the low level routines directly. We compute the k and k_i from Equations 2.4 and 2.5 as functions of $\omega_0$for

$$G_p(s) = \frac{1}{(s+1)^5}. \tag{4.1}$$

In the first case we choose constant ζ_0. The curves $k_i(k)$ are shown with solid lines in Figure 4.5. In the second case we choose ζ_0 such that we get a prescribed M_s value for all ω_0. The $k_i(k)$ are shown with dashed lines. ζ_0 is chosen in the interval 0.0:0.1:1, and M_s in the interval 1.5:0.5:3.5. As can be seen a maximum of a dotted line corresponds reasonably well to a maximum of a dashed line. This Matlab computation takes rather long time.

```
clg;
pstr = '1./(s+1).^5'; w0s = [0.01:0.02:1];
z0s = [0:0.1:0.9 0.999]; mss = [1.5:0.5:3.5];
for ix = z0s,
  c = dptable1(pstr, 'pi', w0s, ix);
  hold on; plot(getk(c), getki(c));
end;
axis([-1 3 0 0.7])
for ix = mss,
  c = dptable2(pstr, 'pi', w0s, ix);
  hold on; plot(getk(c), getki(c),'--');
end;
xlabel('k'); ylabel('ki');
```

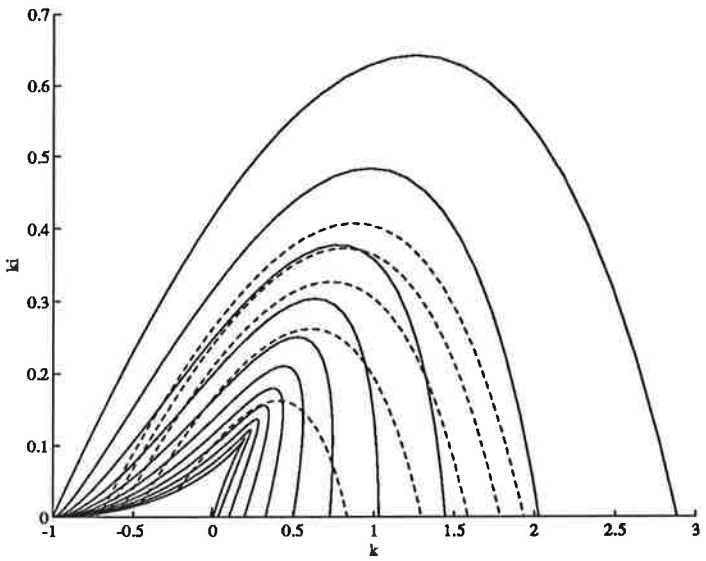

Figure 4.5 Plots of $k_i(k)$ for constant ζ_0 (solid lines) and constant M_s (dashed lines) for varying ω_0.

These kinds of curves are very useful to have to get a feel for how a system react for choices of different ζ_0 and ω_0. □

EXAMPLE 4.7—Interface to Simnon

In some cases it may be desirable to use Simnon for simulation, for example when we want to handle time delays or non-linearities. This example demonstrates how easily it is done with the Simnon interface routines.

The routines 'makep' and 'setsimnon' are designed to accept the same parameters describing a transfer function. This requires that all the necessary systems are implemented in Simnon, or that Matlab can generate the necessary Simnon code. In these examples the Simnon code is written by hand.

In this example we design a PID controller for the process

$$G(s) = \frac{e^{-2s}}{s+1}$$

and simulate the step and load responses for different values of the set point weighting factor β. The plots show a family of step responses when β assumes the values 0:0.2:1.

```
clg; w0s = [0.1:0.1:10]; alpha0 = 1.01; ms = 2.0;
betas = 0:0.2:1; ts = 30;
pstr = makep(1, 1, 2, 1);
con = piddesms(pstr, w0s, ms, alpha0);
setsimnon(1, 1, 2, 1);
par('t1', ts/2);  % t1 is the time when the load starts acting
setpid(con);
for ix = betas,
  par('b', ix);
  y = simu(0, ts);
  subplot(2, 1, 1); hold on; plot(y(:, 1), y(:, 2)); drawnow;
  subplot(2, 1, 2); hold on; plot(y(:, 1), y(:, 3)); drawnow;
end;
subplot(2, 1, 1); grid; xlabel('Time'); ylabel('Output signal');
subplot(2, 1, 2); grid; xlabel('Time'); ylabel('Control signal');
```

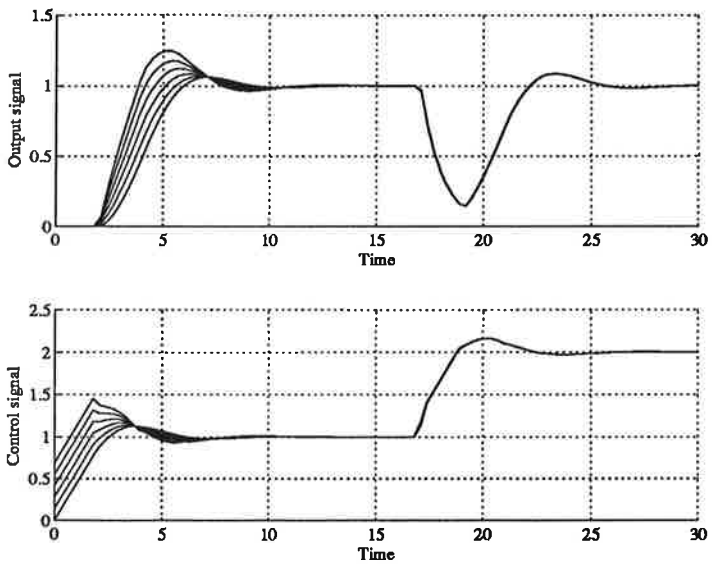

Figure 4.6 Step and load responses for different β in the interval [0 1].

□

5. The Routines

Only a brief description of the routines will be given in this report. The complete source code listing is given in [Persson, 1992a]. The help text of the functions will also explain the meaning of the parameters.

Brief Description of the Routines

Design Routines	
`ides`	I controller with maximal k_i.
`pides`	PI controller with specified ζ_0 and maximal k_i.
`pides2`	PI controller with specified M_s and maximal k_i.
`pidesms`	PI controller with maximal k_i and ζ_0 chosen to get specified M_s.
`pidesam`	PI controller with maximal k_i and ζ_0 chosen to get specified A_m.
`pidespm`	PI controller with maximal k_i and ζ_0 chosen to get specified φ_m.
`betades`	Design of the set point weighting factor.
`piddeskd`	PID controller with specified ζ_0 and k_d and maximal k_i.
`piddes`	PID controller with specified ζ_0 and α_0 and maximal k_i.
`piddes2`	PID controller with specified M_s and α_0 and maximal k_i.
`piddesms`	PID controller with maximal k_i and ζ_0 chosen to get specified M_s.
`piddesam`	PID controller with maximal k_i and ζ_0 chosen to get specified A_m.
`piddespm`	PID controller with maximal k_i and ζ_0 chosen to get specified φ_m.
`piddesm2`	PID controller with maximal k_i and specified M_{s1} and M_{s2}.
`pddes`	PD controller with specified ζ_0 and maximal k.
`pddesms`	PD controller with maximal k and ζ_0 chosen to get specified M_s.
`pddesam`	PD controller with maximal k and ζ_0 chosen to get specified A_m.
`pddespm`	PD controller with maximal k and ζ_0 chosen to get specified φ_m.
`zn1pi`	PI according to Ziegler-Nichols oscillation method.
`zn1pid`	PID according to Ziegler-Nichols oscillation method.
`zn2pi`	PI according to Ziegler-Nichols step response method.
`zn2pid`	PID according to Ziegler-Nichols step response method.

Computation Routines	
`amarg`	Computes the amplitude margin.
`pmarg`	Computes the phase margin.
`asolvecl`	Solves $\lvert L(i\omega)/(1+L(i\omega))\rvert = a$, with $L(i\omega) = G_c(i\omega)G_p(i\omega)$
`asolveol`	Solves $\lvert G_c(i\omega)G_p(i\omega)\rvert = a$.
`psolvecl`	Solves $\arg L(i\omega)/(1+L(i\omega)) = \alpha$, with $L(i\omega) = G_c(i\omega)G_p(i\omega)$
`psolveol`	Solves $\arg G_c(i\omega)G_p(i\omega) = \alpha$.
`mpbeta`	Finds the M_p value of the closed system as function of β.
`mscl`	Finds the M_s value.
`mshelp`	Help routine used in '`piddesm2`'.
`sfrcol`	Finds the frequency responses. For Bode and Nyquist plots.

Basic Pole Placement Routines	
`dptable1`	Interface to the pole placement routines. Argument $\omega_0\zeta_0\alpha_0$.
`dptable2`	Interface to the pole placement routines. Argument $\omega_0 M_s\alpha_0$.
`dptable3`	Interface to the pole placement routines. Argument $\omega_0 A_m\alpha_0$.
`dptable4`	Interface to the pole placement routines. Argument $\omega_0\varphi_m\alpha_0$.
`dpi`	Pole placement of an I controller.
`dpp`	Pole placement of an P controller.
`dppd`	Pole placement of an PD controller.
`dppi`	Pole placement of an PI controller.
`dppid`	Pole placement of an PID controller.
`dppid2`	Pole placement of an PID controller with specified k_d.

Help Routines	
`closeit`	Computes $x/(1+x)$.
`cols`	Returns the number of columns of a matrix.
`evals`	Evaluates a string for a given value of '`s`'.
`evalx`	Evaluates a string for a given value of '`x`'.
`kdguess`	A guess of the maximal k_d. Used in '`piddesm2`'.
`locmax`	Finds a local maximum in an array.
`phase`	Computes the phase of a vector of complex numbers.
`plotc`	Plots a circle.
`rows`	Returns the number of rows of a matrix.

Default Handling Routines	
`defprint`	Controls the printout form some of the routines.
`deftol`	Gives the tolerance for the optimization and solving routines.

Optimization and Solving Routines	
`opt`	One parameter optimizer.
`optg`	Computes the global optimum on an interval. Used in '`mscl`'.
`solve`	One parameter equation solver.
`solveb`	Solving routine used in '`psolve*`', to get the phase right. Inefficient.

Conversion Routines	
`con2str`	Convert the controller matrix to the PID controller string
`convert`	Converts a transfer function string to accept vector arguments.
`geta0`	Access function to get α_0.
`getk`	Access function to get k.
`getkd`	Access function to get k_d.
`getki`	Access function to get k_i.
`gettd`	Access function to get T_d.
`getti`	Access function to get T_i.
`getw0`	Access function to get ω_0.
`getz0`	Access function to get ζ_0.
`makep`	Returns a number of standard systems as strings.
`numtostr`	Converts a numerical value to a string.
`par2con`	Converts PID parameter to the standard controller data structure.
`qstring`	Returns a string with quotes.
`sfun`	Implements the functions of '`makep`'.
`tf2str`	Converts transfer function to a string.

The function calls

An alphabetic list of the function with input and output arguments is also given for reference. The following argument conventions have been used

`am`	=	amplitude margin
`con`	=	the controller data structure (see Section 3.2)
`contype`	=	a string describing the controller type
`cstr`	=	a controller expressed as a string
`k, ti, td`	=	PID controller parameters
`ms`	=	the M_s parameter
`pm`	=	phase margin
`pstr`	=	a transfer function expressed as a string
`str`	=	a string
`tol`	=	the tolerance for equation solving and optimization
`ws, w0, w0s`	=	an array of frequencies
`z0`	=	the ζ_0 parameter
`zguess`	=	changes default value of ζ_0 search interval

The design routines return the controller parameters k, T_i, and T_d. If only one output argument is present in a call of the design functions, the complete controller structure is returned. For a complete description of the functions, use the help texts of the functions. Parameters which have default values are written in *italics*.

```
[am, wx]  = amarg(cstr, pstr, ws, tol)
      wx  = asolvecl(cstr, pstr, y, ws, tol)
      wx  = asolveol(cstr, pstr, y, ws, tol)
       b  = betades(pstr, con, mp, tol)
     res  = closeit(f)
     res  = cols(matrix)
     str  = con2str(k, ti, td, n)
     str  = convert(pstr)
      dp  = defprint
      dt  = deftol
     con  = dpi(pstr, w0)
     con  = dpp(pstr, w0)
     con  = dppd(pstr, w0, z0)
     con  = dppi(pstr, w0, z0)
     con  = dppid(pstr, w0, z0, alpha0)
     con  = dppid2(pstr, w0, z0, kd0)
     con  = dptable1(pstr, contype, w0s, z0, p1)
     con  = dptable2(pstr, contype, w0s, ms, p1, tol)
     con  = dptable3(pstr, contype, w0s, pm, p1, tol)
     con  = dptable4(pstr, contype, w0s, am, p1, tol)
       r  = evals(str, s)
       r  = evalx(str, x)
      a0  = geta0(con)
       k  = getk(con)
      kd  = getkd(con)
      ki  = getki(con)
```

```
        td  = gettd(con)
        ti  = getti(con)
        w0  = getw0(con)
        z0  = getz0(con)
       con  = ides(pstr, w0s, tol)
        kd  = kdguess(pstr, w0s, z0)
       res  = locmax(array)
       str  = makep(sysnr, p1, p2, p3, p4)
       mpr  = mpbeta(pstr, k, ti, td, betas, tol)
  [ms, ws]  = mscl(cstr, pstr, wx, tol)
       str  = numtostr(num, n)
 [xsol, fx] = opt(str, x0s, tol)
 [xsol, fx] = optg(str, x0s, tol)
       con  = par2con(k, ti, td)
    [k, td] = pddes(pstr, w0s, z0, tol)
    [k, td] = pddesam(pstr, w0s, am, zguess, tol)
    [k, td] = pddesms(pstr, w0s, ms, zguess, tol)
    [k, td] = pddespm(pstr, w0s, pm, zguess, tol)
       phi  = phase(g)
[k, ti, td] = piddes(pstr, w0s, z0, alpha0, tol)
[k, ti, td] = piddes2(pstr, w0s, ms, alpha0, tol)
[k, ti, td] = pidesam(pstr, w0s, am, alpha0, zguess, tol)
[k, ti, td] = piddeskd(pstr, w0s, z0, kd0, tol)
      cons  = piddesm2(pstr, w0s, ms1, ms2, kdx, zguess, tol)
[k, ti, td] = piddesms(pstr, w0s, ms, alpha0, zguess, tol)
[k, ti, td] = pidespm(pstr, w0s, pm, alpha0, zguess, tol)
    [k, ti] = pides(pstr, w0s, z0, tol)
    [k, ti] = pides2(pstr, w0s, ms, tol)
    [k, ti] = pidesam(pstr, w0s, am, zguess, tol)
    [k, ti] = pidesms(pstr, w0s, ms, zguess, tol)
    [k, ti] = pidespm(pstr, w0s, pm, zguess, tol)
              plotc(z, rad, phi1, phi2, lt, ddeg)
  [pm, wx]  = pmarg(cstr, pstr, ws, tol)
       wx   = psolvecl(cstr, pstr, y, ws, tol)
       wx   = psolveol(cstr, pstr, y, ws, tol)
       str  = qstring(str)
       res  = rows(matrix)
        fr  = sfrcol(cstr, pstr, w1, w2, n)
        xx  = solve(str, y, x0, fol, tol)
        xx  = solveb(str, y, x0, fol, tol)
       str  = tf2str(num, den)
    [k, ti] = zn1pi(pstr, w0s, tol)
[k, ti, td] = zn1pid(pstr, w0s, tol)
    [k, ti] = zn2pi(kp, l, t)
[k, ti, td] = zn2pid(kp, l, t)
```

Interface to Simnon

Matlab is excellent for many kinds of numerical computations, but there exists better tools for simulation of dynamical systems. When the systems contain non-linearities or time delays Matlab is normally not adequate. The simulation

program Simnon is then a better suited program. Simnon is described in [SSPA, 1990].

A facility has been written to run simulations in Simnon from Matlab, with Simnon working as a 'computation engine.' With this software it is possible to issue Simnon commands from Matlab, to set Simnon parameters from Matlab, and to get numerical results back from Simnon to Matlab.

It is now described how to use the interface functions. Matlab and Simnon communicate via named Unix-pipes. In the following we assume that Matlab will be run from the directory **/home/nisse/matlabdir** and that Simnon will be run from the directory **/home/nisse/simnondir**. The file **pipsimu** is located in the directory **/home/nisse/lib**.

First we have to make two named pipes in Unix, the pipes will be called ipipe and opipe, and will appear to the user as two files in the directory /home/nisse/simnondir. The named pipes are created with the following commands

```
% /usr/etc/mknod /home/nisse/simnondir/ipipe p
% /usr/etc/mknod /home/nisse/simnondir/opipe p
```

Open a new window, and give the following commands

```
% cd /home/nisse/simnondir
% /home/nisse/lib/pipsimu ipipe
```

This will start Simnon in the window and Simnon will now accept commands from the named pipe ipipe.

The program pipsimu is a Perl script, which starts Simnon, reads commands from ipipe and sends them along to Simnon. This seems unnecessary, but Simnon refuses to read commands directly from a named pipe. For readings on Perl, see [Wall and Schwartz, 1991]. The Perl script is quite short and simple:

```
#!/usr/local/bin/perl
$pipename = shift;
open(SIMNON, "|simnon") || die "cannot pipe\n";
select(SIMNON);
$|=1;
print "x\n";
#print "algor rkf45\n";
while(open(PIPE, $pipename)){
  while(<PIPE>){
    if (eof(PIPE)) {close(PIPE);}
    print;
    if ($_ eq "stop\n") {exit;};
  }
}
```

Now open a window where Matlab will be run, and give the following commands

```
% cd /home/nisse/matlabdir
% matlab
>> global inpipe1 outpipe1 simdir
>> inpipe1  = '/home/nisse/simnondir/ipipe';
```

```
>> outpipe1 = '/home/nisse/simnondir/opipe';
>> simdir   = '/home/nisse/simnondir/';
```

Two Matlab variables 'inpipe1' and 'outpipe1' are now defined and have values of the named pipes. The variables must have the full path names. Simnon will now accept commands from Matlab. A typical command sequence will be

```
>> gp = makep(2, 8);                    % Define a transfer function
>> cp = pidesms(gp, [0.1:0.1:5], 2);    % Design a controller
>> setsimnon(2, 8);                     % Compile the correct model
>> setpid(cp);                          % Set the PID parameters
>> r = simu(0, 100);                    % Simulate
>> plot(r(:,1), r(:, 2));               % Plot the results
```

The functions 'makep' and 'setsimnon' must be written such that the parameters give Simnon commands to compile the correct model files. The function 'setsimnon' typically contains a number of Simnon commands as 'syst' and 'par'.

Listing of functions for interfacing Simnon

In this section a few of the commands for the Matlab-Simnon interface will be listed. Some of these functions must be modified to suit the the Simnon models with which will be used.

The basic command is 'remcommand' which sends text strings to a named pipe. The function 'simc' calls 'remcommand' but uses 'inpipe1' as default for the named pipe, 'simc' also accepts several input parameters.

```
function remcommand(com, inpipe)

%REMCOMMAND  Sends a command string to be executed in a remote
%            process. The command is sent via the named pipe inpipe.

%Copyright (c) 1992 by Per Persson, Department of Automatic Control,
%Lund Institute of Technology, Lund, Sweden
%
%LastEditDate : Tue Dec  8 13:41:16 1992

eval(['!/usr/5bin/echo ' '''' com '''' ' >> ' inpipe]);
```

```
function simc(p1, p2, p3, p4, p5, p6, p7, p8, p9, p10)

%SIMC  A number of text strings are sent to a remote process (Simnon).
%      The command is sent via a named pipe which name is in the
%      global variable inpipe1.
%
%      SIMC(p1)
%      .
%      .
%      .
%      SIMC(p1, p2, p3, p4, p5, p6, p7, p8, p9, p10)

%Copyright (c) 1992 by Per Persson, Department of Automatic Control,
%Lund Institute of Technology, Lund, Sweden
%
%LastEditDate : Tue Dec  8 13:41:16 1992
```

```
global inpipe1
for ix=1:nargin,
  remcommand(eval(sprintf('p%g', ix)), inpipe1);
end;
```

The commands 'getsimval' and 'getsimvec' transfer Simnon variables, parameters and simulation results back to Matlab. 'getsimval' gets the value of a variable or parameter, and 'getsimvec gets the simulation result stored in the file 'store.d' or any other file the user specifies. The Matlab user must know what has been stored in 'store.d' and in which order the signals are stored.

The communication goes via the named pipe 'outpipe1' to get synchronization of Matlab and Simnon. This way Matlab does not go on computing until Simnon has provided the required value.

```
function res = getsimval(val, outpipe)

%GETSIMVAL  Gets a value of a parameter or variable form Simnon.
%
%           res = getsimval(val)
%           val = the name of the parameter or variable as a string
%
%           The value is sent back via a named pipe, outpipe, to
%           synchronize the two processes.

%Copyright (c) 1992 by Per Persson, Department of Automatic Control,
%Lund Institute of Technology, Lund, Sweden
%
%LastEditDate : Sat Dec  5 16:29:50 1992

if nargin==1
  global outpipe1
  outpipe = outpipe1;
end;

simc(['disp ' val '/zztmp']);
simc('write (dk zzfile FF)');
simc('write (dk zzfile) zztmp.');
simc(['$cat zzfile.t >>' outpipe]);
delete('zzfile.txt');
eval(['!cat < ' outpipe ' > zzfile.txt']);
load('zzfile.txt');
res = zzfile;
```

```
function res = getsimvec(file)

%GETSIMVEC  Gets the simulation values from a file stored during a
%           Simnon simulation.
%
%           res = getsimvec(file)
%           file = the store file where the values are stored.
%                  (default: 'store')

%Copyright (c) 1992 by Per Persson, Department of Automatic Control,
%Lund Institute of Technology, Lund, Sweden
%
%LastEditDate : Thu Jan  7 11:42:29 1993
```

```
if nargin==0, file = 'store'; end;
global outpipe1

simc('write (dk zzfile FF)');
simc(['export zzfile <' file ' /0']);
simc(['$cat zzfile.t >> ' outpipe1]);
delete('zzfile.txt');
eval(['!cat < ' outpipe1 ' > zzfile.txt']);
load('zzfile.txt');
res = zzfile;
```

The following two commands implement the Simnon commands 'simu' and 'par' in Matlab. If 'simu' has an output argument then a 'getsimvec' is automatically given to get the simulation result back to Matlab.

```
function res = simu(t0, t1, dt)

%SIMU  Makes the simu command in simnon. The simulation result is
%      returned if there is an out parameter.
%
%      y = SIMU(t0, t1, dt)

%Copyright (c) 1992 by Per Persson, Department of Automatic Control,
%Lund Institute of Technology, Lund, Sweden
%
%LastEditDate : Sat Dec  5 16:29:50 1992

if nargin==2,
  par('dt[logger]', (t1-t0)/100);
  simc(['simu ' num2str(t0) ' ' num2str(t1)]);
else
  par('dt[logger]', dt);
  simc(['simu ' num2str(t0) ' ' num2str(t1) ' ' num2str(dt)]);
end;
if nargout==1, res = getsimvec; end;
```

```
function par(p1, v1, p2, v2, p3, v3, p4, v4, p5, v5)

%PAR  Set parameters in Simnon. The parameter name is given as a
%     string and its value as a number.
%
%     PAR(p1, v1)
%      .
%      .
%      .
%     PAR(p1, v1, p2, v2, p3, v3, p4, v4, p5, v5)
%     pi = the parameter name expressed as a string
%     vi = the numerical value of the parameter

%Copyright (c) 1992 by Per Persson, Department of Automatic Control,
%Lund Institute of Technology, Lund, Sweden
%
%LastEditDate : Tue Dec  8 13:39:30 1992

for ix=1:(nargin/2),
  tmp1 = eval(sprintf('p%g', ix));
  tmp2 = numtostr(eval(sprintf('v%g', ix)));
  simc(['par ' tmp1 ' : ' tmp2]);
```

```
end;
```

Finally, a command for transferring the standard controller data structure to the PID controller parameters in Simnon. This function is only given as an example, in general a function like this is very dependent on how the controller is implemented in Simnon.

```
function setpid(k, ti, td, p4, p5, p6, p7, p8, p9, p10, p11)

%SETPID  Set the PID controller parameters in the controller in
%        Simnon.
%
%        SETPID(con)
%        SETPID(k, ti)
%        SETPID(k, ti, td)

%Copyright (c) 1992 by Per Persson, Department of Automatic Control,
%Lund Institute of Technology, Lund, Sweden
%
%LastEditDate : Sat Dec  5 16:30:53 1992

global simdir
if rows(k)>1,
  disp('More than one row in the gain vector, using the last.');
  k = k(rows(k), :);
end;
if nargin==1 & (cols(k) > 1),
  tmp = k;
  k = getk(tmp);
  ki = getki(tmp);
  ti = getti(tmp);
  td = gettd(tmp);
elseif nargin==2,
  td = 0;
end;

mn = -1e4; mx = 1e4;
k = max(mn, min(k, mx));

fn = [simdir 'pargen.t'];
delete(fn);
fd = fopen(fn, 'w');
fprintf(fd, 'macro pargen\n');

kcp = 1;
if ti==Inf | ki == 0, ti = 1; kci = 0; else kci = 1; end;
if td==0, kcd = 0; td = 1; else kcd = 1; end;

fn = [simdir 'pargen.t'];
delete(fn);
fd = fopen(fn, 'w');
fprintf(fd, 'macro pargen\n');
fprintf(fd, 'par k: %g\n', k);
fprintf(fd, 'par kcp: %g\n', kcp);
fprintf(fd, 'par ti: %g\n', ti);
fprintf(fd, 'par kci: %g\n', kci);
fprintf(fd, 'par kcd: %g\n', kcd);
fprintf(fd, 'par td: %g\n', td);
```

```
for ix=4:2:nargin,
  fprintf(fd, ['par ' eval(sprintf('p%g', ix)) ' : ' ...
               num2str(eval(sprintf('p%g', ix+1))) '\n']);
end;
fprintf(fd, 'end\n');
fclose(fd);
simc('pargen');
```

6. Conclusions

The Matlab toolbox presented in this report implements a simple and systematic tuning procedure for PID controllers, with a good design variable. The design routines are modular and has proven to be easy to extend to implement new design ideas.

The design is built on a string representation of transfer functions, and con in principle handle any transfer function. All optimization and equation solution can be carried out with an arbitrary precision.

The design method computes a controller which is a compromise between robustness (the condition on M_s) and performance (the maximization of k_i). The method can handle a broad spectrum of dynamics uniformly in the sense that the time and frequency responses of the controlled systems will look very similar. It is, however, difficult to investigate the closed loop systems obtained with the method analytically, except in the simplest cases.

Systems with poorly damped poles

Some care must be taken if the design method presented is used for systems with poorly damped poles. The principle to maximize k_i for a given value od M_s is still a fruitful one, but the routines '`pidesms`' and '`piddesms`' cannot be used for systems with poorly damped poles. It may work for PID controllers, but certainly not for PI controllers. The reason is that the approximations for finding ζ_0 are not longer valid. Instead we must use the general routines in '`dptable2`', but this gives rise to several computational problems. It is also important to choose the design parameter α_0 properly for oscillating systems, $\alpha_0 < 0.5$. These problems are currently (December 1992) topics for further research.

7. References

GUSTAFSSON, K., M. LILJA, and M. LUNDH (1990a): "A collection of Matlab routines for control system analysis and synthesis." Internal Report TFRT-7454, Department of Automatic Control, Lund Institute of Technology.

GUSTAFSSON, K., M. LILJA, and M. LUNDH (1990b): "A collection of Matlab routines for control system analysis and synthesis, the code." Internal Report TFRT-7455, Department of Automatic Control, Lund Institute of Technology.

MATHWORKS (1990): *MATLAB – User's Guide.* MathWorks, Cochituate Place, 24 Prime Park Way, Natick, MA 01760, USA.

PERSSON, P. (1992a): "The dominant pole design toolbox – the Matlab code." Technical Report TFRT-7498, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

PERSSON, P. (1992b): *Towards Autonomous PID control, Part B: PID Controller Design.*, Ph. D. Thesis. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

SSPA (1990): *SIMNON User's Guide for MS-DOS Computers.* SSPA Systems, Box 24001, S-400 22 Götcborg, Sweden. Version 3.0.

WALL, L. and R. SCHWARTZ (1991): *Programming Perl.* O'Reilly & Associates, Inc., Sebastopol, CA.

ZIEGLER, J. G. and N. B. NICHOLS (1942): "Optimum settings for automatic controllers." *Transactions of the ASME*, **64**, pp. 759–768.

ZIEGLER, J. G. and N. B. NICHOLS (1943): "Process lags in automatic-control circuits." *Transactions of the ASME*, **65:5**, pp. 433–443.

ÅSTRÖM, K. J. and T. HÄGGLUND (1988): *Automatic Tuning of PID Controllers.* Instrument Society of America (ISA), Research-Triangle Park, NC.

ÅSTRÖM, K. J. and T. HÄGGLUND (1984): "Automatic tuning of simple regulators with specifications on phase and amplitude margins." *Automatica*, **20:5**, pp. 645–651.