FORTBILDUNGSSEMINAR
“NEUE TECHNOLOGIEN IN DER PROZESSAUTOMATISIERUNG“

Part 1:

Review of classical and modern Control Theory Using Matlab® Student Edition[1]

Prof. Dr. Urban Brunner
TECSUP Peru
September 2000

[1] published by PRENTICE HALL (price: $\leq$ 50 US$)
“Student Versions” of Matlab’s toolboxes are available in USA/Canada by The MathWorks

Index

1. Introduction into Matlab *3*

1.1 Vectors, Matrices, and Polynomials 3

1.2 M-files and Matlab Functions 10

1.3 Macros and Programming Utilities 11

1.4 Functions of Matlab's Student Edition 13

2. Representation of Linear Systems *15*

2.1 State space variables and transfer function 15

2.2 Linear systems transformations 17

2.3 Composition of (linear) systems 18

3. Simulation and Optimization *19*

3.1 Simulation 19

3.2 Runge-Kutta Integration 20

3.3 Optimization 21

4. Classical Control *23*

4.1 Rationale of classical control 23

4.2 PID Controller 24

4.3 Review on classical control 25

5. Modern Control *26*

5.1 Linear Quadratic Control 26

5.2 Pole placement and design of observers 27

5.3 Critique of LQG 28

6. Symbolic computation in control *32*

6.1 Symbolic commands of Matlab's Student Edition 32

6.2 An illustrative example 34

6.3 Sampling of continuous systems 36

1. Introduction into Matlab

Matlab is an interactive program for numerical computation and data visualization; it is used extensively by control engineers for analysis and design. There are many different toolboxes available which extend the basic functions of Matlab into different application areas. For instance, the Matlab toolboxes “Control System”, “Model Predictive Control”, and “Robust Control” provide methods for computer-aided control system design. In this document, we will make use of the student version of Matlab which is available for personal computers. Matlab is supported on Unix, Macintosh, and Windows environment. For more information on Matlab, contact The MathWorks (http://www.mathworks.com).

The idea behind this seminar document is that you can view it in one window while running Matlab in another window. You should be able to re-do all of the plots and calculations in the document by cutting and pasting text from the document into Matlab CommandWindow or an m-file.

1.1 Vectors, Matrices, and Polynomials

Vectors

Let's start off by creating something simple, like a vector. Enter each element of the vector (separated by a space) between brackets, and set it equal to a variable. For example, to create the vector a, enter into the Matlab CommandWindow (you can "copy" and "paste" from here into Matlab to make it easy):

```
a = [1 2 3 4 5 6 9 8 7]
```

Matlab should return:

```
a =
   1  2  3  4  5  6  9  8  7
```

The elements of an vector are referenced by indexing; e.g. a(1)=1, a(9)=7. Actually, a vector is an one-dimensional array which is the basic data element of Matlab. In Matlab an array does not require dimensioning. Let's say you want to create a vector with elements between 0 and 20 evenly spaced in increments of 2 (this method is frequently used to create a time vector):

```
t = 0:2:20
```

```
t =
   0  2  4  6  8  10  12  14  16  18  20
```

Manipulating vectors is almost as easy as creating them. First, suppose you would like to add 2 to each of the elements in vector 'a'. The equation for that looks like:

```
b = a + 2
```

```
b =
   3  4  5  6  7  8  11  10  9
```

Now suppose, you would like to add two vectors together. If the two vectors are the same length, it is easy. Simply add the two as shown below:

```
c = a + b

    c =
        4  6  8  10  12  14  20  18  16
```

Subtraction of vectors of the same length works exactly the same way.

Built-in functions

To make life easier, Matlab includes many standard functions; so called built-in functions or commands. Each function is a block of code that accomplishes a specific task. Matlab contains all of the standard functions such as sin, cos, log, exp, sqrt, as well as many others. Commonly used constants such as pi, and i or j for the square root of -1, are also incorporated into Matlab.

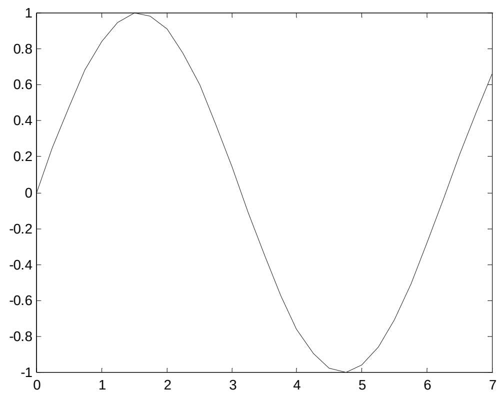
```
sin(pi/4)

ans =

    0.7071
```

To determine the usage of any function, type help [functionname] at the Matlab command window. Matlab even allows you to write your own functions with the function command. See chapter 1.2 to learn how to write your own functions.

Plotting

It is also easy to create plots in Matlab. Suppose you wanted to plot a sine wave as a function of time. First make a time vector (the semicolon after each statement tells Matlab we don't want to see all the values) and then compute the sin value at each time.

```
t=0:0.25:7;
y = sin(t);
plot(t,y)
```



The plot contains approximately one period of a sine wave. Basic plotting is very easy in Matlab, and the plot command has extensive add-on capabilities. (To learn more about plotting type help plot in the Matlab command window.)

Polynomials

In Matlab, a polynomial is represented by a vector. To create a polynomial in Matlab, simply enter each coefficient of the polynomial into the vector in descending order. For instance, let's say you have the following polynomial:

$$s^4 + 3s^3 - 15s^2 - 2s + 9$$

To enter this into Matlab, just enter it as a vector in the following manner

```
x = [1 3 -15 -2 9]

    x =
       1  3  -15  -2  9
```

Matlab can interpret a vector of length n+1 as an nth order polynomial. Thus, if your polynomial is missing any coefficients, you must enter zeros in the appropriate place in the vector. For example,

$$s^4 + 1$$

would be represented in Matlab as:

```
y = [1 0 0 0 1]
```

You can find the value of a polynomial using the polyval function. For example, to find the value of the above polynomial at s=2,

```
z = polyval([1 0 0 0 1],2)

    z =
       17
```

You can also extract the roots of a polynomial. This is useful when you have a high-order polynomial such as

$$s^4 + 3s^3 - 15s^2 - 2s + 9$$

Finding the roots would be as easy as entering the following command;

```
roots([1 3 -15 -2 9])


    ans =
      -5.5745
       2.5836
      -0.7951
       0.7860
```

Let's say you want to multiply two polynomials together. The product of two polynomials is found by taking the convolution of their coefficients. Matlab's function conv will do this for you.

```
x = [1 2];
y = [1 4 8];
z = conv(x,y)
```

```
z =
   1  6  16  16
```

Dividing two polynomials is just as easy. The deconv function will return the remainder as well as the result. Let’s divide z by y and see if we get x.

```
[xx, R] = deconv(z,y)

   xx =
      1  2

   R =
      0  0  0  0
```

As you can see, this is just the polynomial/vector x from before. If y had not gone into z evenly, the remainder vector would have been something other than zero.

If you want to add two polynomials together which have the same order, a simple z=x+y will work (the vectors x and y must have the same length). In the general case, the user-defined function, polyadd can be used. To use polyadd, copy the function into an m-file, and then use it just as you would any other function in the Matlab toolbox. Assuming you had the polyadd function stored as a m-file, and you wanted to add the two uneven polynomials, x and y, you could accomplish this by entering the command:

```
function[poly]=polyadd(poly1,poly2)
%polyadd(poly1,poly2) adds two polynominals possibly of uneven length
if length(poly1)<length(poly2)
  short=poly1;
  long=poly2;
else
  short=poly2;
  long=poly1;
end
mz=length(long)-length(short);
if mz>0
  poly=[zeros(1,mz),short]+long;
else
  poly=long+short;
end
```

```
z = polyadd(x,y)

   x =
      1  2

   y =
      1  4  8

   z =
      1   5  10
```

Matrices

Entering matrices into Matlab is the same as entering a vector, except each row of elements is separated by a semicolon ‘;’ or a return:

```
B = [1 2 3 4;5 6 7 8;9 10 11 12]

   B =
      1    2    3    4
      5    6    7    8
      9   10   11   12
```

```
B = [ 1  2  3  4
     5  6  7  8
     9 10 11 12]
```

```
B =
     1    2    3     4
     5    6    7     8
     9   10   11    12
```

Matrices in Matlab can be manipulated in many ways. For one, you can find the transpose of a matrix using the apostrophe key:

```
C = B'
```

```
C =
     1    5     9
     2    6    10
     3    7    11
     4    8    12
```

It should be noted that if C had been complex, the apostrophe would have actually given the complex conjugate transpose. Now you can multiply the two matrices B and C together. Remember that order matters when multiplying matrices.

```
D = B * C
```

```
D =
    30     70    110
    70    174    278
   110    278    446
```

```
D  = C * B
```

```
D =
   107    122    137    152
   122    140    158    176
   137    158    179    200
   152    176    200    224
```

Another option for matrix manipulation is that you can multiply the corresponding elements of two matrices using the '.*' operator (the matrices must be the same size to do this).

```
E = [1 2;3 4]
F = [2 3;4 5]
G = E .* F
```

```
E =
    1   2
    3   4

F =
    2   3
    4   5

G =
    2   6
   12  20
```

If you have a square matrix, like E, you can also multiply it by itself as many times as you like by raising it to a given power.

```
E^3

ans =
   37    54
   81   118
```

If wanted to cube each element in the matrix, just use the element-by-element cubing.

```
E.^3

ans =
    1     8
   27    64
```

You can also find the inverse of a matrix:

```
X = inv(E)

X =
  -2.0000    1.0000
   1.5000   -0.5000
```

or its eigenvalues:

```
eig(E)

ans =
  -0.3723
   5.3723
```

There is even a function to find the coefficients of the characteristic polynomial of a matrix. The "poly" function creates a vector that includes the coefficients of the characteristic polynomial.

```
p = poly(E)

p =

   1.0000   -5.0000   -2.0000
```

Remember that the eigenvalues of a matrix are the same as the roots of its characteristic polynomial:

```
roots(p)

    ans =
       5.3723
      -0.3723
```

Printing

Printing in Matlab is pretty easy. To print a plot or a m-file from a computer running Windows, just select Print from the File menu in the window of the plot or m-file, and hit return.

Getting help in Matlab

Matlab has a fairly good on-line help; type

```
help commandname
```

for more information on any given command. You do need to know the name of the command that you are looking for.

Here are a few notes to end this chapter.

You can get the value of a particular variable at any time by typing its name.

```
B
     B =
          1   2   3
          4   5   6
          7   8   9
```

You can also have more that one statement on a single line, so long as you separate them with either a semicolon or comma. Also, you may have noticed that so long as you don't assign a variable to the result of an operation, Matlab will store it in a temporary variable called "ans". By the way, Matlab is case sensitive.

File management

To create a record of all inputs and outputs in a Matlab session, and to save the record under a given name in ASCII format, use the diary command: diary filename.

The command save filename saves the data to a file with the given name in the binary Matlab format. If no extension is specified, Matlab adds the extension .mat to the filename. Note that this command saves the workspace (variables you have defined or results of operations), while diary saves input and output records only and not the actual data. You can also perform a partial save; i.e., save some of the data. For example,

```
save filename   x  y  z       %  saves variables x, y, and z to file
save filename1   x  /ascii  /double      %  saves variable x in 16-digit ASCII format
```

1.2 M-files and Matlab Functions

What is an m-file?

An m-file, or script file, is a simple text file where you can place Matlab commands. When the file is run, Matlab reads the commands and executes them exactly as it would if you had typed each command sequentially at the Matlab prompt. All m-file names must end with the extension '.m' (e.g. plot.m). If you create a new m-file with the same name as an existing m-file, Matlab will choose the one which appears first in the path order (help path for more information). To make life easier, choose a name for your m-file which doesn't already exist. To see if a filename.m exists, type help filename at the Matlab prompt.

Why use m-files?

For simple problems, entering your requests at the Matlab prompt is fast and efficient. However, as the number of commands increases or trial and error is done by changing certain variables or values, typing the commands over and over at the Matlab prompt becomes tedious. M-files will be helpful and almost necessary in these cases.

How to create, save or open an m-file?

If you are using PC to create an m-file, choose New from the File menu and select m-file. This procedure brings up a text editor window in which you can enter Matlab commands. To save the m-file, simply go to the File menu and choose Save (remember to save it with the '.m' extension). To open an existing m-file, go to the File menu and choose Open .

How to run the m-file?

After the m-file is saved with the name filename.m in the Matlab folder or directory, you can execute the commands in the m-file by simply typing filename at the Matlab prompt. If you don't want to run the whole m-file, you can just copy the part of m-file that you want to run and paste it at the Matlab prompt.

Matlab Functions

When entering a command such as roots, plot, or step in Matlab what you are really doing is running an m-file with inputs and outputs that has been written to accomplish a specific task. These types of m-files are similar to subroutines in programming languages in that they have inputs (parameters which are passed to the m-file), outputs (values which are returned from the m-file), and a body of commands which can contain local variables. Matlab calls these m-files functions. You can write your own functions using the function command.

The new function must be given a filename with a '.m' extension. This file should be saved in the same directory as the Matlab software, or in a directory which is contained in Matlab's search path. The first line of the file should contain the syntax for this function in the form:

```
function [output1,output2] = filename(input1,input2,input3)
```

A function can input or output as many variables as are needed. The next few lines contain the text that will appear when the help filename command is evoked. These lines are optional, but must be entered using % in front of each line in the same way that you include comments in an ordinary m-file. Finally, below the help text, the actual text of the function with all of the commands is included. One suggestion would be to start with the line:

```
error(nargchk(x,y,nargin));
```

The x and y represent the smallest and largest number of inputs that can be accepted by the function; if more or less inputs are entered, an error is triggered.

Functions can be rather tricky to write, and practice will be necessary to successfully write one that will achieve the desired goal. Below is a simple example of what the function, add.m, might look like.

```
function [var3] = add(var1,var2)
%add is a function that adds two numbers
var3 = var1+var2;
```

If you save these three lines in a file called "add.m" in the Matlab directory, then you can use it by typing at the command line:

```
y = add(3,8)
```

Obviously, most functions will be more complex than the one demonstrated here. This example just shows what the basic form looks like. Look at the functions created for this tutorial listed below, or at the functions in the toolbox folder in the Matlab software, for more sophisticated examples, or try `help function` for more information.

The following list of commands can be very useful for future reference. Use `help` in Matlab for more information on how to use the commands. In this document, we use commands both from Matlab and from the Control Systems Toolbox, as well as some commands/functions which we wrote ourselves.

1.3 Macros and Programming Utilities

Macros are a short sequence of Matlab commands that are used often in a session. They can be saved for use in future as part of other m-files. Macros are created as text strings and are executed using the `eval` command. The next example uses a macro to compute factorials:

```
 fct='prod(1:n)' ;
```

We will use the macro to find 10!

```
n=10; eval(fct)

ans =

    3628800
```

Matlab has several commands that allow you to make your programs interactive, and help with debugging your programs. We will briefly discuss some of these commands.

```
pause
```

stops an m-file until any key is pressed. It is useful after plotting commands; `pause(n)` will pause for n seconds and `pause(-2)` cancels all subsequent pauses.

```
echo    or    echo on     or    echo off
```

displays the program contents while an m-file is being executed. It is a toggle command, so typing echo by itself will change the echo state. If echo is on, it affects all script files; it affects function files differently. (For details enter help echo.)

```
keyboard
```

stops an m-file and gives the control to the keyboard. You can view and change all variables as you wish. Typing return and pressing the return (or enter) key allows the m-file to continue. It is useful for debugging programs.

```
x=input('prompt')
```

The input command displays the text string as a prompt, waits for the user to enter a response, evaluates the expression entered by the user and assigns it to x. It is the input command that lets you create interactive programs. For example, we now write an interactive factorial macro:

```
f1='prod(1:n)' ;
f2='n=input(''enter number:__''); z=eval(f1); disp(''n! is''), disp(z)'  ;
eval(f2)

        enter number:__5
        n! is
          120
```

Note the following points. The interactive macro, f2, can call another macro, f1. When you use strings containing apostrophes ('), you must double the apostrophes (do not use quotation marks). For instance, the f2 macro is enclosed by apostrophes; the strings within the input and disp commands are also enclosed by apostrophes; the apostrophes for these strings are doubled.

```
exist('item')
```

Checks for the existence of a variable in the workspace or a file in the Matlab's path. It returns 1 if the variable exists; it returns 2 if the file exists; it returns 0 otherwise.

```
path
```

path, by itself, prettyprints Matlab's current search path. path(path, P) appends a new directory to the current path; for instance, path(path,'c:\matlab\work\goodstuff').

1.4 Functions of Matlab's Student Edition

Command	**Description**
abs	Absolute value
acker	Compute the K matrix to place the poles of A-BK, see also place
axis	Set the scale of the current plot, see also plot, figure
bode	Draw the Bode plot, see also logspace, margin, nyquist1
c2dm	Continuous system to discrete system
clf	Clear figure
conv	Convolution (useful for multiplying polynomials), see also deconv
ctrb	The controllability matrix, see also obsv
deconv	Deconvolution and polynomial division, see also conv
det	Find the determinant of a matrix
dimpulse	Impulse response of discrete-time linear systems, see also dstep
dlqr	Linear-quadratic requlator design for discrete-time systems, see also lqr
dlsim	Simulation of discrete-time linear systems, see also lsim
dstep	Step response of discrete-time linear systems, see also stairs
eig	Compute the eigenvalues of a matrix
eps	Matlab's numerical tolerance
feedback	Feedback connection of two systems
figure	Create a new figure or redefine the current figure, see also subplot, axis
for	For, next loop
format	Number format (significant digits, exponents)
function	Creates function m-files
grid	Draw the grid lines on the current plot
gtext	Add a piece of text to the current plot, see also text
help	HELP!
hold	Hold the current graph, see also figure
if	Conditionally execute statements
imag	Returns the imaginary part of a complex number, see also real
impulse	Impulse response of continuous-time linear systems, see also step, lsim, dlsim
input	Prompt for user input
inv	Find the inverse of a matrix
length	Length of a vector, see also size
linspace	Returns a linearly spaced vector
log	natural logarithm, also log10: common logarithm
loglog	Plot using log-log scale, also semilogx/semilogy
logspace	Returns a logarithmically spaced vector
lqr	Linear quadratic regulator design for continuous systems, see also dlqr
lsim	Simulate a linear system, see also step, impulse, dlsim.
margin	Returns the gain margin, phase margin, and crossover frequencies, see also bode
norm	Norm of a vector
nyquist	Draw the Nyquist plot.

Matlab functions (cont.)

obsv	The observability matrix, see also ctrb
ones	Returns a vector or matrix of ones, see also zeros
place	Compute the K matrix to place the poles of A-BK, see also acker
plot	Draw a plot, see also figure, axis, subplot.
poly	Returns the characteristic polynomial, roots to polynomial conversion
polyadd	Add two different polynomials
polyval	Polynomial evaluation
print	Print the current plot (to a printer or postscript file)
pzmap	Pole-zero map of linear systems
rank	Find the number of linearly independent rows or columns of a matrix
real	Returns the real part of a complex number, see also imag
residue	Partial fraction expansion
rlocfind	Find the value of k and the poles at the selected point
rlocus	Draw the root locus
roots	Find the roots of a polynomial
rscale	Find the scale factor for a full-state feedback system
set	Set(gca,'Xtick',xticks,'Ytick',yticks) to control the number and spacing of tick marks on the axes
series	Series interconnection of Linear time-independent systems
size	Gives the dimension of a vector or matrix, see also length
sqrt	Square root
ss	Create state-space models or convert LTI model to state space, see also tf
ss2tf	State-space to transfer function representation, see also tf2ss
ss2zp	State-space to pole-zero representation, see also zp2ss
stairs	Stairstep plot for discreste response, see also dstep
step	Plot the step response, see also impulse, lsim, dlsim.
subplot	Divide the plot window up into pieces, see also plot, figure
text	Add a piece of text to the current plot, see also title, xlabel, ylabel, gtext
tf	Creation of transfer functions or conversion to transfer function, see also ss
tf2ss	Transfer function to state-space representation, see also ss2tf
tf2zp	Transfer function to Pole-zero representation, see also zp2tf
title	Add a title to the current plot
xlabel/ylabel	Add a label to the horizontal/vertical axis of the current plot, see also title, text
zeros	Returns a vector or matrix of zeros
zgrid	Generates grid lines of constant damping ratio (zeta) and natural frequency (Wn), see also sgrid, jgrid, sigrid
zp2ss	Pole-zero to state-space representation, see also ss2zp
zp2tf	Pole-zero to transfer function representation, see also tf2zp

2. Representation of Linear Systems

2.1 State space variables and transfer function

We start from the differential equations of a linear (second order) system. Introducing (two) “state space variables” x_1 and x_2 the differential equations can be written down in the following normalized form the so called “state space representation” of a system:

$$\underbrace{\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix}}_{\underline{\dot{x}}} = \underbrace{\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}}_{A} * \underbrace{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}}_{\underline{x}} + \underbrace{\begin{bmatrix} b_1 \\ b_2 \end{bmatrix}}_{\underline{b}} * u$$

$$\underline{\dot{x}}(t) = A\underline{x}(t) + \underline{b}u(t) \quad ; \qquad \underline{x}(0) = \underline{x}_0$$

$$y = \underline{c}^T \underline{x}(t) + du(t)$$

The well known solution consists of two parts; one caused by the initial condition $\underline{x}_0$ and the other one by the input signal u(t):

$$y(t) = \underline{c}^T e^{At} \underline{x}_0 + \int_0^t \underline{c}^T e^{A(t-\tau)} \underline{b}u(\tau)d\tau + du(t)$$

where

$$e^{at} = 1 + at + a^2 \frac{t^2}{2} + a^3 \frac{t^3}{3!} + ...$$

$$e^{At} = I + At + A^2 \frac{t^2}{2} + A^3 \frac{t^3}{3!} + ..$$

Using Laplace transforms the state space equations can be written as

$$s\underline{X}(s) - \underline{x}_0 = A\underline{X}(s) + \underline{b}U(s)$$

$$Y(s) = \underline{c}^T \underline{X}(s) + dU(s)$$

and we get

$$Y(s) = \underline{c}^T (sI - A)^{-1} \underline{x}_0 + \underline{c}^T (sI - A)^{-1} \underline{b}U(s) + dU(s)$$

The transfer function G(s) of a SISO-system is defined by:

$$Y(s) = G(s)U(s)\Big|_{\underline{x}_0 = \underline{0}}$$

Thus, we have

$$G(s) = \underline{c}^T (sI - A)^{-1} \underline{b} + d = \frac{\underline{c}^T adj(sI - A)\underline{b}}{\det(sI - A)} + d = \frac{z(s)}{n(s)}$$

The denominator n(s) is the characteristic polynomial of the system and has degree n = dim($\underline{x}$). Here, we have assumed that there are no common roots in n(s) and z(s). Often, the system has no direct path from the input u(t) to the output y(t); i.e. d = 0 . In this case the nominator z(s) has degree $m \leq n-1$. The following two forms of G(s) :are commonly used:

$$G(s) = \frac{z(s)}{n(s)} = \frac{b_m s^m + b_{m-1} s^{m-1} + \ldots + b_1 s + b_0}{s^n + a_{n-1} s^{n-1} + \ldots + a_1 s + a_0}$$

respectively.: $$G(s) = \frac{z(s)}{n(s)} = k \frac{(s - s_1)(s - s_2)(s - s_3)\ldots(s - s_m)}{(s - p_1)(s - p_2)(s - p_3)\ldots(s - s_n)}$$

In Matlab polynomials are represented by vectors (see page 5) and consequently, a transfer function by two vectors. In contrast, a state space representation of a system is given by four matrices A, B, C, D and possibly an initial condition:

$$\begin{aligned} \dot{\underline{x}}(t) &= A\underline{x}(t) + B\underline{u}(t) \ ; \qquad \underline{x}(0) = \underline{x}_0 \\ \underline{y}(t) &= C\underline{x}(t) + D\underline{u}(t) \end{aligned}$$

The interpretation/meaning of the matrices is given by the order they have been entered. Thereby, the dimensions of the matrices have to be noted.

```
A1=[0 1; -2 –3];
b1=[0;1]; % column vector
c1=[1 0]; % row vector
d1=0; % or any scalar
```

Using the class constructor ss you get an object that represents the system.

```
sys=ss(A1,b1,c1,d1)

a =
             x1      x2
      x1      0       1
      x2     -2      -3

b =
             u1
      x1      0
      x2      1

c =
             x1      x2
      y1      1       0

d =
             u1
      y1      0

Continuous-time model.
```

The corresponding matrix is referenced by its field-index. For instance, you get the dynamic matrix by

```
sys.a

ans =

     0     1
    -2    -3
```

2.2 Linear systems transformations

By means of the following functions the representation of a system can be altered:

ss2tf	—	state space to transfer function conversion
ss2zp	—	state space to zero-pole conversion
tf2ss	—	transfer function to state space conversion
tf2zp	—	transfer function to zero-pole conversion
zp2ss	—	zero-pole to state space conversion
zp2tf	—	zero-pole to transfer function conversion

Examples:

```
[z,n]=ss2tf(sys.a,sys.b,sys.c, sys.d)   %  or simply   [z,n]=ss2tf(A1,b1,c1,d1)

z =

     0     0     1


n =

     1     3     2
```

```
[z,p,k]=tf2zp(num,den)     %   converts a transfer function given in the first form into the second form
```

2.3 Composition of (linear) systems

Usually, systems can be thought of interconnected sub-systems often represented as interconnected blocks. The block diagram of a standard control loop is shown in fig.1.

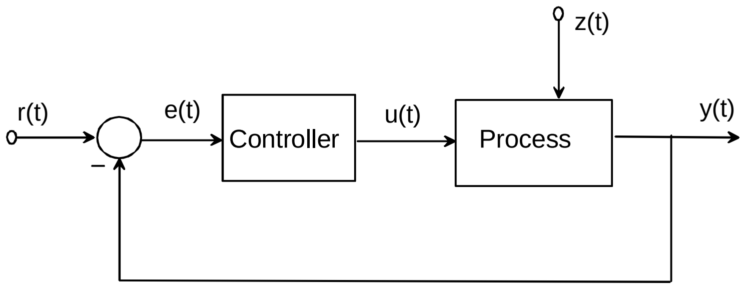


Fig.1: Standard control loop

Using the function append two systems can be concatenated to one, and by ssselect a subsystem can be extracted from a larger system. The following Matlab code shows the application of series and cloop to compute the closed-loop transfer function.

```
roots=[ -1  -1  -1];  %  poles of the process
nProc=[1] ;  dProc=poly(roots) ;  %  nominator and denominator of the process
Kp=1;  Ki=1;  %  PI-controller with corresponding gains
nCon=[Kp  Ki]; dCon=[1  0];  %  nominator and denominator of the controller
[nopen, dopen]=series(nCon, dCon, nProc, dProc)   %  loop gain
[nCloop, dCloop]=cloop(nopen, dopen, -1)  % closed loop transfer function
```

```
nopen =

     0     0     0     1     1

dopen =

     1     3     3     1     0

nCloop =

     0     0     0     1     1

dCloop =

     1     3     3     2     1
```

3. Simulation and Optimization

3.1 Simulation

Here, simulation should be understood as the computation of the numerical solution of ordinary differential equations (ode). Using ode23 or ode45 differential equations of the form

$$\dot{\underline{x}}(t) = \underline{f}\,(\underline{x}(t), u(t), t);\;\; \underline{x}(0) = \underline{x}_0$$

are numerically integrated/simulated. Linear systems can be simulated also using the function lsim;

```
[y,x]=lsim(A,b,c,d,u,t,x0)
```

As lsim uses a corresponding linear time discrete system for integration the vector t has to be linearly spaced. The functions ode23 and ode45 are more general. They have to be called in the form

```
[t,x]=ode23('funct', t0 tf, x0)
```

where t is a vector containing the time points, x is the state vector, t0 is the start time, tf is the end (final) time, and x0 is the initial condition. 'funct' is the name of a m-function describing the model equations. These equations have the form xDot=funct(tnow, x) where tnow is a scalar representing the actual time. For example, a second order system excited by a harmonic input signal can be modeled and simulated by the following Matlab code:

```
t0=0; tf=10;  x0=[0;0];
[t,x]=ode45('SystemOrd2', t0, tf, x0);
plot(t,x); hold on;
plot(t, sin(t)); hold off;
```

The m-file 'SystemOrd2' contains the model equations:

```
function xdot=SystemOrd2(t, x);
% second order system excited by a harmonic input signal
% file SystemOrd2.m
xdot=[x(2); -x(1)-x(2)+sin(t)];
```

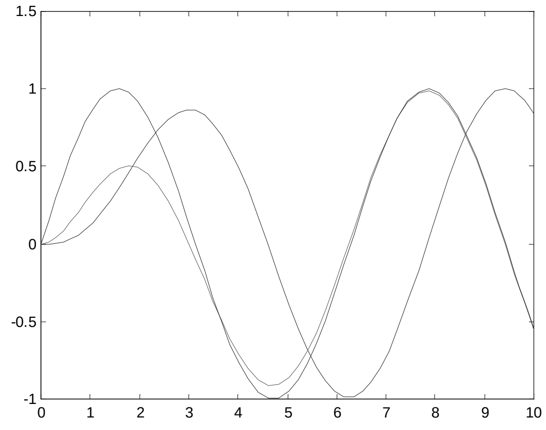


Fig. 2: Simulation results

ode45 adapts automatically the integration step in order to achieve the specified accuracy with less computing effort. But in some cases (e.g. for identification and optimization), it is useful to have an integration algorithm with fixed step. In the next chapter, a Runge-Kutta algorithm with fixed integration step is shown..

3.2 Runge-Kutta Integration

From textbooks you get the following formulas

$$\underline{k}_1(t) = \underline{f}(\underline{x}_k, t_k);$$

$$\underline{k}_2(t) = \underline{f}(\underline{x}_k + \frac{h}{2}\underline{k}_1, t_k + \frac{h}{2});$$

$$\underline{k}_3(t) = \underline{f}(\underline{x}_k + \frac{h}{2}\underline{k}_2, t_k + \frac{h}{2});$$

$$\underline{k}_4(t) = \underline{f}(\underline{x}_k + h\underline{k}_3, t_k + h);$$

$$\underline{x}_{k+1} = \underline{x}_k + \frac{h}{6}\left(\underline{k}_1 + 2k_2 + 2k_3 + k_4\right);$$

The following Matlab function ‘rk4Abcd’ provides the above mentioned Runge-Kutta integration of a linear system described by A,b,c,d. The meaning of the Input parameters is obvious: A,b,c,d x0 (initial condition), u (control input), t0 , tf , dt (fixed integration step).

```
function y=rk4Abcd(A,b,c,d,x0,u,t0,tf,dt);
% file rk4Abcd.m
% runge kutta integration of state space equations
d1=1/6; i=0;
t=t0;
y=[];
x=x0;
%
while  t < tf ,
   tn=t; xn=x; i=i+1;
   xdot=A*x + b*u(i);
   c1=dt*xdot;
   t= tn + 0.5*dt; x= xn + 0.5*c1;
   c2=dt*xdot;
   x=xn+ 0.5*c2;
   xdot=A*x+b*u(i) ;
   c3=dt*xdot ;
   t= tn +dt;  x=xn+c3 ;
   xdot=A*x + b*u(i) ;
   c4= dt*xdot ;
   x = xn + d1*(c1 + 2.*c2 + 2.*c3 + c4);
   y = [y, c*x+d*u(i)] ;
   end;
```

3.3 Optimization

For the multi-dimensional optimization the Matlab student edition provides the Matlab function fmins. – Of course, the Optimization Toolbox offers a bundle of specific optimization methods.– fmins is called by

fmins('f', p) , where f is the function to be minimized with respect to the parameter vector p .

Let’s optimize a two term controller on a third order plant using fmins. As the function f to be minimized, an usual quadratic performance index has been chosen. To prevent the parameters of the PI-controller to become negative values the term exp(-10*p)'*exp(-10*p) has been added. Apart from that, the following m-files are almost self-explanatory. Common (global) variables have to be declared by the (reserved) keyword global.

```
%  optimizing a classical two term controller
global qmin popt
hold off;
qmin=10000;
popt=[] ;
p=[]; p(1,1)=1 ; p(2,1)= 1 ;
D=eye(2);
epsi=0.000001 ;
fmins('twotermpi',p)
```

```
function q=twotermpi(p);
% performance index for two term controller optimization
% on a third order plant
global qmin popt
kp=p(1);  ki=p(2);
A=[-1 1 0 0; 0 -1 1 0; -kp 0 -1 1; -ki 0 0 0];
b=[0; 0;  kp; ki];
c=[1 0 0 0; -kp 0 0 1];
d=[0; kp];
x0=[0; 0; 0; 0];
t0=0;  tf=20; dt=0.2;
u=ones((tf-t0)/dt+1,1);
y=rk4Abcd(A,b,c,d,x0,u,t0,tf,dt);
[n1,n2]=size(y);
yr=ones(n1,n2);
e=(y-yr)*(y-yr)' ;
q=e(1,1)+e(2,2)+exp(-10*p)'*exp(-10*p);
if  q < qmin,  qmin=q, plot(y(1,:), '-w'); popt=[popt, p];  hold on;  end ;
```

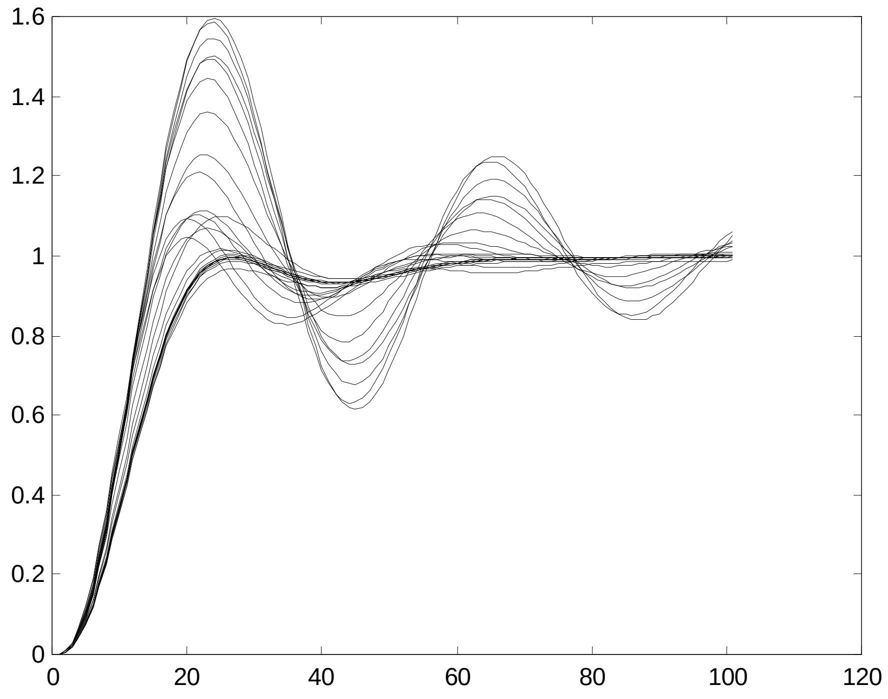


Fig. 3: Improving of the step response by the optimization

```
hold off
j= 1:1:length(popt(1,:));
plot(j,popt)
```

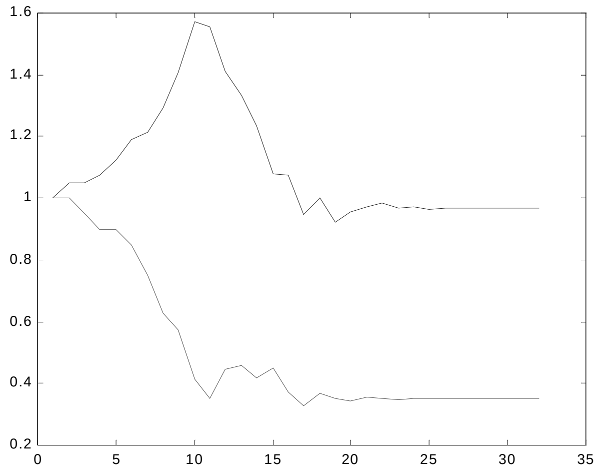


Fig. 4: Convergence of the parameters kp and ki (above: kp)

4. Classical Control

4.1 Rationale of classical control

Classical control can be best understood by considering the so called “servo dilemma” (see fig. 3).

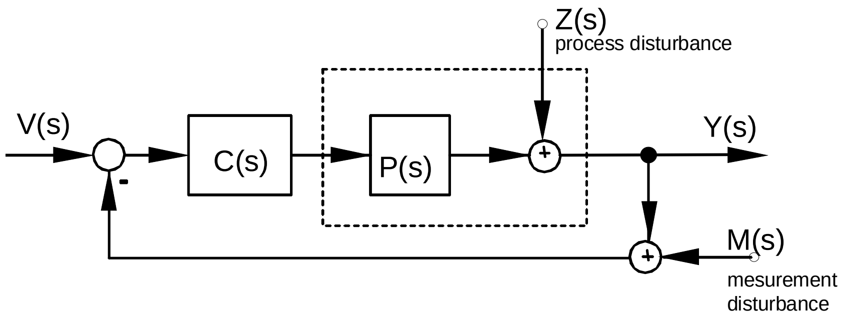


Fig. 5: standard control loop with (unknown) process and measurement disturbances

By applying the superposition principle to this control loop the output signal y(t) and the corresponding transfer functions can be obtained:

$$Y(s) = G_V(s) * V(s) + G_Z(s) * Z(s) + G_M(s) * M(s)$$

with the (open) loop gain $G(s) = C(s) * P(s)$:

Design goals	Transfer function	Specifications for the closed loop	Resulting specs. for the loop gain G(s)
Good tracking	$G_V(s) = \frac{G(s)}{1+\mathrm{G(s)}}$	$G_V(s) \approx 1$	$G(s) >> 1$
Disturbance rejection (“Sensitivity”)	$G_Z(s) = \frac{1}{1+\mathrm{G(s)}}$	$G_Z(s) \approx 0$	$G(s) >> 1$
Noise suppression	$G_M(s) = \frac{-G(s)}{1+\mathrm{G(s)}}$	$G_M(s) \approx 0$	$G(s) << 1$

Thus, in a standard control configuration the following relations hold independently of the controller C(s):

(i) $G_V(s) + G_Z(s) = 1$ (That’s why $G_V(s)$ is called “Complementary Sensitivity”.)

(ii) $\| G_V(s) \| = \| G_M(s) \|$

In other words, you cannot achieve simultaneously these three goals. Consequently, a trade-off between “Disturbance rejection” and “Noise suppression” has to be found. The servo dilemma is usually resolved by assuming that there is no noise in the low frequency range. In particular, it is supposed that the sensors signals are not corrupted by an offset nor by a drift. In this case, the loop gain G(s) should exhibit the following shape:

$\Vert G(j\omega) \Vert$	$>>$ 1	in the frequency range: $\omega << \omega_c$
$\Vert G(j\omega_c) \Vert$	$=$ 1	ω_c : so called. „Crossover frequency“
$\Vert G(j\omega) \Vert$	$<<$ 1	for $\omega >> \omega_c$

From the specified shape of the loop gain G(s) the transfer function of the controller C(s) can be obtained by an “inverse” operation.

4.2 PID Controller

The transfer function of a PID (three term) controller is as follows (cf. fig. 6)

$$C(s) = k_P + \frac{k_I}{s} + k_D s = \frac{k_D s^2 + k_P s + k_I}{s} \quad , \textit{where} \quad \begin{array}{l} k_P \text{ : Proportional gain} \\ k_I \text{ : Integral gain} \\ k_D \text{ : Derivative gain} \end{array}$$

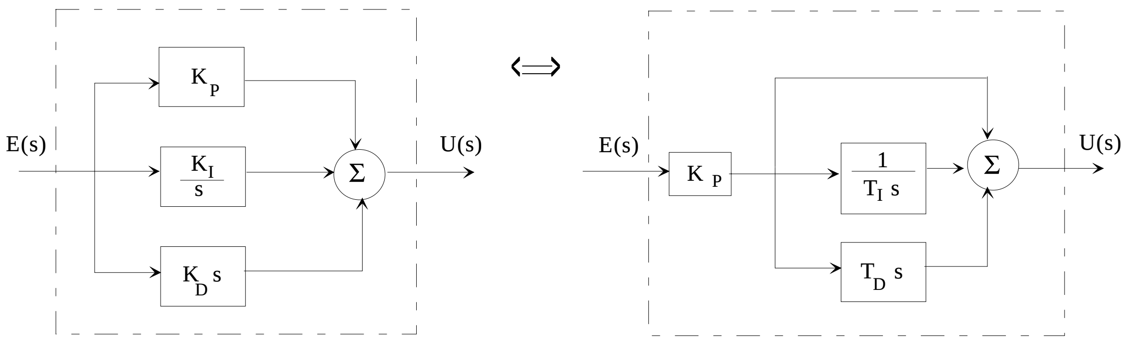


Fig. 6: Commonly used block diagram representations / parameterizations of a PID controller

A proportional controller (K_P) will have the effect of reducing the rise time and will reduce, but never eliminate, the steady-state error. An integral control (K_I) will have the effect of eliminating the steady-state error, but it may make the transient response worse. Integral action is often used for compensating constant (but unknown) process disturbances; e.g. to compensate the offset of an actuator for instance the leakage of a servo valve. A derivative control (K_D) will have the effect of increasing the stability of the system, reducing the overshoot, and improving the transient response.

When you are designing a PID controller for a given system, follow the steps shown below to obtain a desired response.

1. Obtain an open-loop response and determine what needs to be improved
2. Add a proportional control to improve the rise time
3. Add a derivative control to improve the overshoot
4. Add an integral control to eliminate the steady-state error
5. Adjust each of K_P, K_I, and K_D until you obtain a desired overall response.

For analysing the stability/robustness and the time response the following functions are available

Time response:

impulse	– Impulse response of continuous-time linear systems
lsim	– Continuous simulation to arbitrary inputs
step	– Step response

Frequency response:

bode	– Bode plot
nyquist	– Nyquist plot
margin	– Returns the gain margin, phase margin, and crossover frequencies
rlocus	– Evans root locus

4.3 Review on classical control

Using feedback a lot of plants will behave very well. In particular, for (almost) linear, stable, minimum-phase processes of rather low order P, PI, and PID controllers can be successfully designed by classical methods. [Un-]fortunately, the real world is not linear – life would be unendurably boring. But, classical control methods cannot cope with (hard) non-linearities. Moreover, a constraint on the control input caused by saturation of an actuator can result in an unacceptable overshooting when using integral action without an adequate countermeasure. – The reader is invited to show this overshooting (so called “Windup phenomenon”) by simulation. – The windup of the integrator can be prevented by holding on the integrator while the saturation is active or by a reset of the integrator. According to the actual operating condition the integrator is switched off and on respectively. Today, using “fuzzy logic” a smooth switching can be achieved. Indeed, the combination of classical control and fuzzy control allows to tackle the difficulties in controlling non-linear and/or time-variant processes. As the Matlab student edition doesn’t support fuzzy control, these issues are not further discussed in this document.

5. Modern Control

5.1 Linear Quadratic Control

The notion of state suggests to use the information of all state variables for computing the (optimal) control input whereas classical control uses output and maybe its derivative.

Consider the linear system and the quadratic cost function *J*

$$\dot{\underline{x}}(t) = A\underline{x}(t) + B\underline{u}(t) \; ; \qquad \underline{x}(0) = \underline{x}_0$$
$$\underline{y} = C\underline{x}(t)$$
$$J = \frac{1}{2}\int_0^T \left(\underline{x^T}(t)Q\underline{x}(t) + \underline{u}^T(t)R\underline{u}(t)\right)dt$$

The problem is to minimize *J* with respect to the control input $\underline{u}(t)$. This is known as the linear quadratic regulator (LQR) problem. A simple interpretation of the cost function is as follows. If the system is first order, the cost function becomes

$$J = \frac{1}{2}\int_0^T \left(q\,x^2(t) + r\,u^2(t) \right)dt$$

Now we see that *J* represents the weighted sum of energy of the state and control. If *r* is very large relative to *q* , the control energy is penalized very heavily. This physically translates into smaller motors, actuators, and amplifier gains needed to implement the control law. Likewise if *q* is much larger than *r* , the state is penalized heavily, resulting in a very damped system. Note that we require that *Q* be symmetric positive semi-definite (written as $Q \geq 0$) and *R* symmetric positive definite ($R>0$) for a meaningful optimisation problem.

Optimal control problems can be solved using variation calculus. For obvious reasons we omit here the derivation of the result:

$$-\frac{dP}{dT} = A^T P + PA + Q - PBR^{-1}B^T P \; ; \qquad P(T) = 0$$

The above is the famous Riccati differential equation. It is a nonlinear first order differential equation that has to be solved backwards in time. The above formulation and solution of the LQR problem is known as the finite time (or finite horizon) problem. It results in a linear time varying controller of the feedback form

$$\underline{u}(t) = -K(t)\underline{x}(t) \quad where \qquad K(t) = R^{-1}B^T P(t)$$

For the infinite time LQR problem, we let *T* approach infinity. Of course, now one runs into the question of the convergence of the cost function and, hence, the existence of the optimal controller. It turns out that under mild conditions, *P(t)* approaches a constant matrix *P* (hence $dP/dt \to 0$), and the positive definite solution of the algebraic Riccati equation results in an asymptotically stable closed loop system.

$$0 = A^T P + PA + Q - PBR^{-1}B^T P$$
$$\underline{u}(t) = -K\underline{x}(t) \quad where \qquad K = R^{-1}B^T P$$

The command lqr solves the (infinite time) LQR problem directly. The syntax is given by

```
[K,P,ev]= lqr(A,B,Q,R,N)      %   where N=0 is most commonly used
```

5.2 Pole placement and design of observers

An alternative to LQR is pole placement whereby the feedback matrix K is computed such that arbitrary desired eigenvalues/poles result. In the pole-placement approach to the design of control systems, we assume that all state variables are available for feedback. In practice, however, this is not so. Then we need to estimate unavailable state variables. It is important to note that we should avoid differentiating a state variable to generate another one. Differentiation of a signal always decreases the signal-to-noise ratio because noise generally fluctuates more rapidly than the command signal. A device that estimates the state variables is called an observer. Basically, an observer is a parallel model of the plant which is driven by the same input like the plant (see fig. 7). By comparing the actual plant output with the estimated output the observer follows the plant. The coupling between plant and observer is given by a feedback matrix L Roughly spoken, a strong coupling is needed when process disturbances and/or model uncertainties are present whereas a measurement noise would indicate a rather weak coupling.

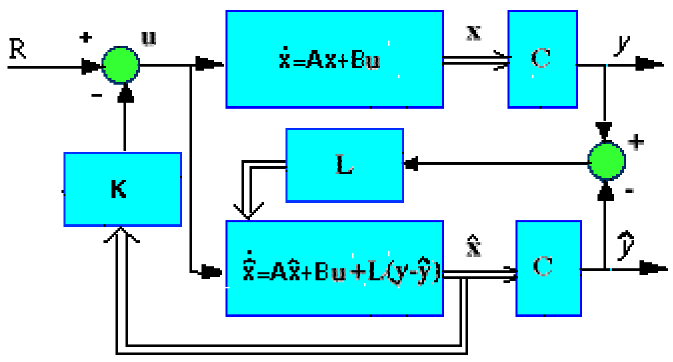


Fig. 7: Control architecture pole placement with observer

It can be shown that the design of the feedback K and the design of the (observer) feedback L are basically the same problems; they are called “dual”. For the gain selection the following commands are provided:

acker	– SISO pole placement
lqe	– Linear-quadratic estimator design (Kalman-Bucy Filter gains)
lqr	– Linear-quadratic regulator design
place	– Pole placement

5.3 Critique of LQG

LQR has many desirable properties. Among them are good stability margins and sensitivity properties what will be shown next by an example. The numerical results and plots (figures 8, 9, 10, and 11) are generated by running the following m-script:

```
% Design of a servo system
hold off
% representation of the servo system
A=[0 1 0 0
   -42 -1.5 42 0
   0 0 0 1
   42 0 -42 0];
b=[0 3.8*pi 0 0]';
c=[0 0 1/pi 0];
d=0;
% design of the controller
r=1;
Q=c'*c;  % Anderson
[k,S,E]=lqr(A,b,Q,r);
k
E
step(A-b*k,b,c,0,1); pause;
%
w=1.1:0.02:50;
[num,den]=ss2tf(A,b,k,0,1);
[Re, Im]=nyquist(num,den,w);
t=0:0.1:6.3; plot(-1+sin(t),cos(t),'k'); hold on;
plot(Re,Im);axis([-2 1 -1.5 1.5]);grid;
hold off; pause;
% design of the observer
rho=2.e3;
Q=b*b'*(1+rho);
[h,S,E]=lqr(A',c',Q,1);
h
E
step(A-h'*c,b,c,0,1); pause;
Ac=[A, b*k;0*eye(4), A-b*k-h'*c];
bc=[0 0 0 0 h]';
cc=[c 0 0 0 0];
w=0.8:0.03:100;
[Reo,Imo]=nyquist(Ac,bc,cc,0,1,w);
t=0:0.1:6.3; plot(-1+sin(t),cos(t),'k'); hold on;
plot(Reo,Imo);axis([-2 1 -1.5 1.5]);grid;
hold off;
```

```
k =

   0.2554    0.1164    0.0629    0.1134


E =

  -0.3865 + 9.1298i
  -0.3865 - 9.1298i
  -1.0582 + 0.8897i
  -1.0582 - 0.8897i
```

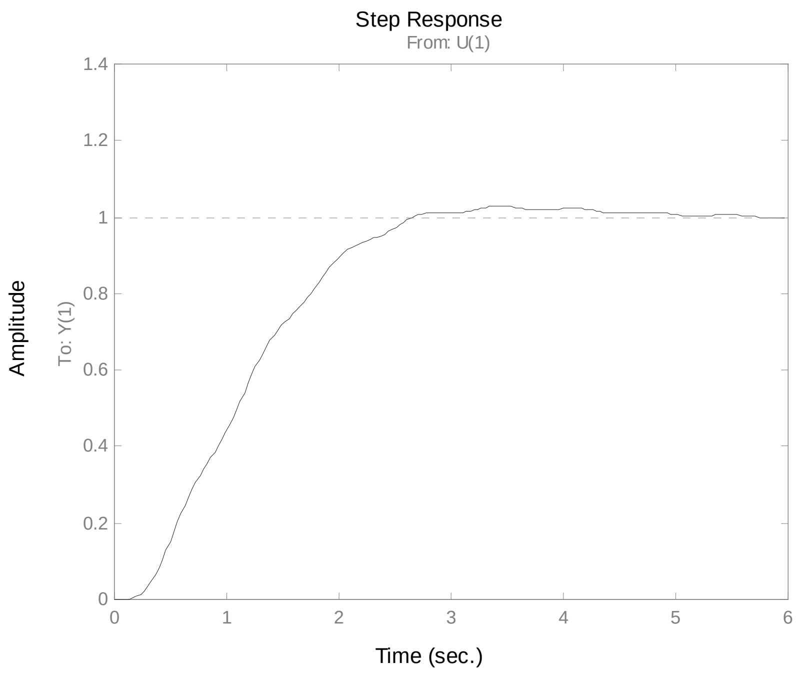


Fig. 8: Step response of the controlled system

The rather slow design (r=1) results in a slow step response. Fig. 9 shows the resulting Nyquist plot. The Nyquist plot clearly stays out of the unit disc with centre at (-1,0) what ensures a stability phase margin of ≥ 60 degrees. This is the desired and famous property of LQR design; also known as “Kalman-Yakubovich inequalities”.

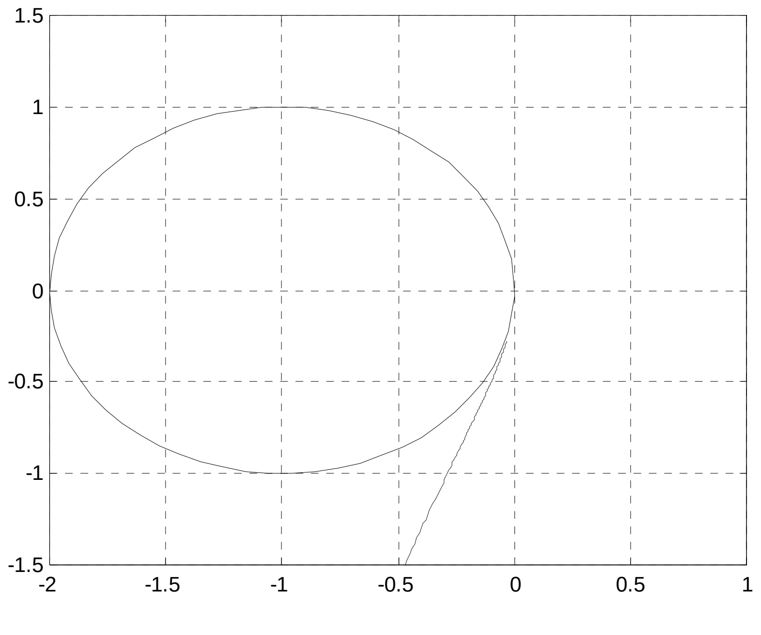


Fig. 9: Nyquist plot of the controlled system without observer

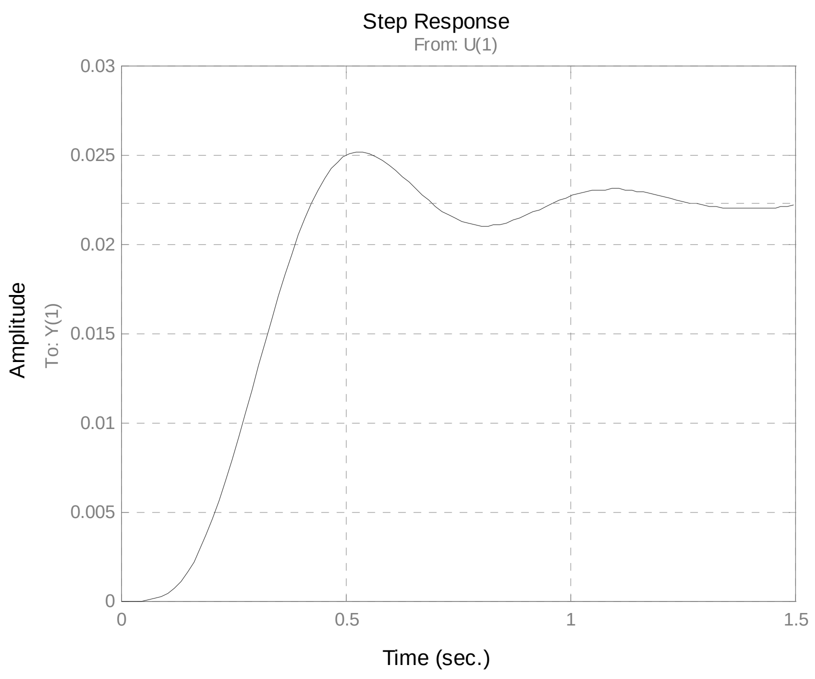


Fig. 10: "Step response" of the observer / error dynamics

With rho=2000 , i.e. process disturbances have to be more considered than measurement noise, a rather strong coupling and relatively fast estimation error dynamics are obtained:

```
h =

   68.1236    4.6087   51.8106  427.2253

E =

  -6.5139 + 4.0644i
  -6.5139 - 4.0644i
  -2.4820 +10.7213i
  -2.4820 -10.7213i
```

Fig. 11 shows the Nyquist plot of the resulting loop gain when using an observer. It must be noted that the robustness property of LQR is lost by using an observer. The Nyquist plot stays no longer out of the unit disc with centre (-1,0). The major problem with LQG-based design is the lack of robustness. Furthermore, it can be shown that LQG-based design could become unstable in practice as more realism was added to the plant model. It became apparent that too much emphasis on optimality, and not enough attention to the model uncertainty issue, was the main cause for the failing of LQG control. During the eighties, much of the attention was shifted back to feedback properties and frequency-domain techniques. In particular, the H_∞-technique should be mentioned. Today, the Matlab toolboxes "Robust Control", and "μ-Analysis and Synthesis" exist for robust controller design. By the way, why are the step responses of the closed loop system with and without observer identical?

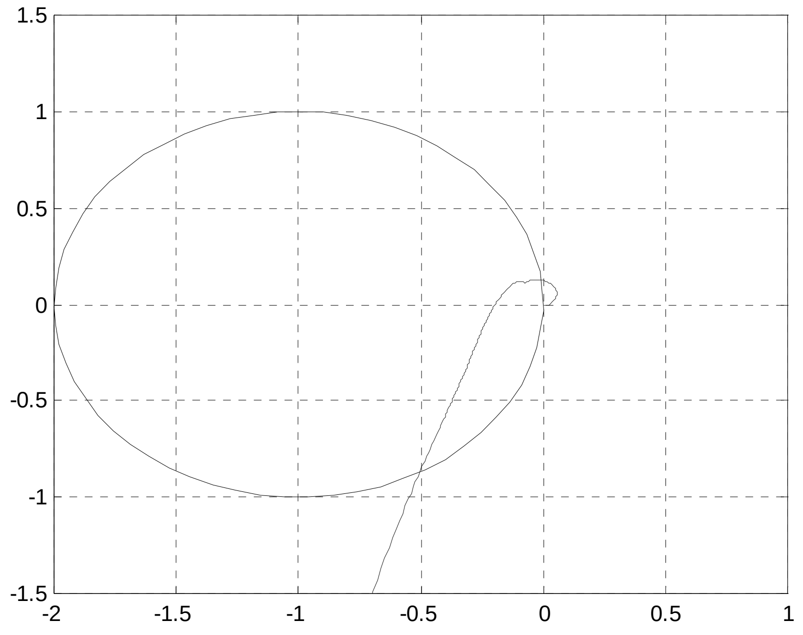
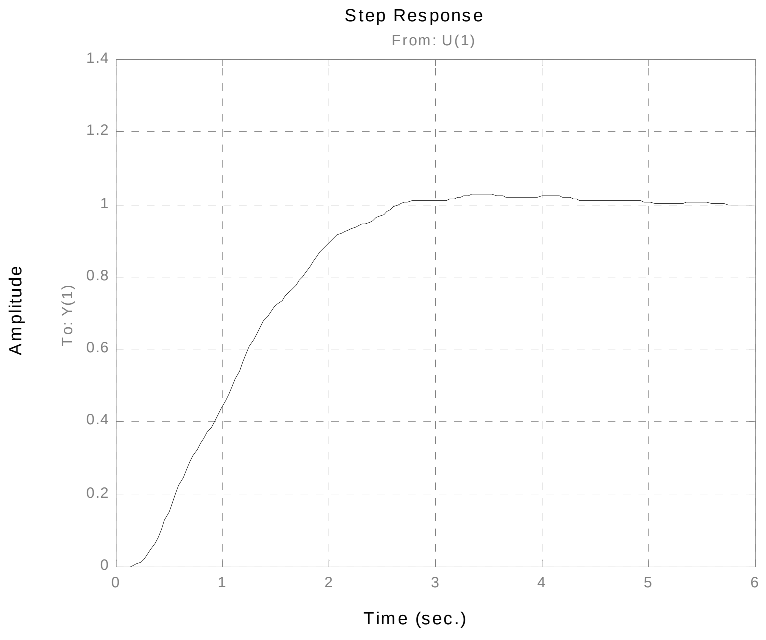


Fig. 11: Nyquist plot of the controlled system with observer

```
% check the step response of the
% closed loop system with observer

bn=[b' b']';
cn=[c 0 0 0 0];
An=[A, -b*k; h'*c, A-b*k-h'*c];
step(An,bn,cn,0,1)

eig(An)
```

```
ans =

  -2.4820 +10.7213i
  -2.4820 -10.7213i
  -0.3865 + 9.1298i
  -0.3865 - 9.1298i
  -6.5139 + 4.0644i
  -6.5139 - 4.0644i
  -1.0582 + 0.8897i
  -1.0582 - 0.8897i
```



6. Symbolic computation in control

6.1 Symbolic commands of Matlab's Student Edition

Beside the numerical methods which we have made use of as far, techniques based on symbolic computation are increasingly used in the field of automatic control. An introduction into symbolic computation and its object-oriented realization in Matlab would go beyond the scope of this seminar. Rather, we like to show the use of symbolic commands by discussing some standard problems whereby we restrict on commands of the Matlab Student Edition which are a selection/subset of Maple® commands.

```
help symbolic
```

```
 Symbolic Math Toolbox.
 Version 1.1    (Student Edition)

 Calculus.
   diff         -  Differentiate.
   int          -  Integrate.
   taylor       -  Taylor series.
   jacobian     -  Jacobian matrix.

 Linear Algebra.
   inverse      -  Symbolic matrix inverse.
   determ       -  Symbolic matrix determinant.
   linsolve     -  Solve simultaneous linear equations
   nullspace    -  Basis for null space.
   colspace     -  Basis for column space.
   eigensys     -  Symbolic eigenvalues and eigenvectors.
    transpose   -  Symbolic matrix transpose.
   singvals     -  Singular values and singular vectors.
   jordan       -  Jordan canonical (normal) form.
   charpoly     -  Symbolic characteristic polynomial.

 Simplification.
   simplify     -  Simplify.
   expand       -  Expand.
   factor       -  Factor.
   collect      -  Collect.
   simple       -  Search for shortest form.
   allvalues    -  Find all values for RootOf expression..
   symsum       -  Symbolic summation.

 Solution of Equations.
   solve        -  Symbolic solution of algebraic equations.
   dsolve       -  Symbolic solution of differential equations.
   finverse     -  Functional inverse.
   compose      -  Functional composition.
```

Variable Precision Arithmetic.

vpa	-	Variable precision arithmetic.
digits	-	Set variable precision accuracy.

(Integral) Transforms.

fourier	-	Fourier transform.
laplace	-	Laplace transform.
ztrans	-	Z transform.
invfourier	-	Inverse Fourier transform.
invlaplace	-	Inverse Laplace transform.
invztrans	-	Inverse Z transform.

Conversions.

double	-	Convert symbolic matrix to double.
poly2sym	-	Coefficient vector to symbolic polynomial.
sym2poly	-	Symbolic polynomial to coefficient vector.
char	-	Convert sym object to string.

Operations on Symbolic Expressions and Matrices.

numeric	-	Convert symbolic matrix to numeric form.
sym	-	Create or modify symbolic object.
symvar	-	Determine symbolic variables..
symop	-	Symbolic operations.
symadd	-	Add symbolic expressions.
symsub	-	Subtract symbolic expressions.
symmul	-	Multiply symbolic expressions.
symdiv	-	Divide symbolic expressins.
sympow	-	Power of symbolic expressions.
symsize	-	Size of symbolic expressions.
subs	-	Substitute for subexpression.
numden	-	Numerator and denominator.
poly2sym	-	Coefficient vector to symbolic polynomial.
sym2poly	-	Symbolic polynomial to coefficient vector.
horner	-	Nested polynomial representation.
pretty	-	Pretty print a symbolic expression.
latex	-	LaTeX representation of a symbolic expression.

Pedagogical and Graphical Applications.

rsums	-	Riemann sums..
ezplot	-	Easy to use function plotter.
funtool	-	Function calculator.

Access to Maple. (Not available with Student Edition.)

maple	-	Access Maple kernel.
mfun	-	Numeric evaluation of Maple functions.
mfunlist	-	List of functions for MFUN.
mhelp	-	Maple help.
procread	-	Install a Maple procedure. (Requires Extended Toolbox.)

It should be noticed that some function-calls have been changed with the approach of object-oriented programming style in Matlab 5.1 (see "Symbolic Math Toolbox.Version 2.1.1"). Matlab 5 makes use of operator overloading. For instance, the function `symadd(s1,s2)` is obsolete because `symadd(s1,s2)` is the same as `sym(s1)+sym(s2)` .

6.2 An illustrative example

Using symbolic computation the state space model of a simple RC network and its transfer function will be computed (see fig. 12).

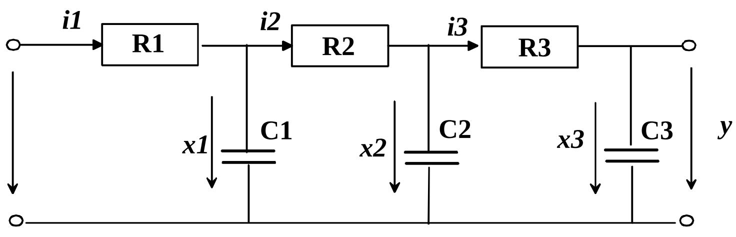


Fig. 12: RC network

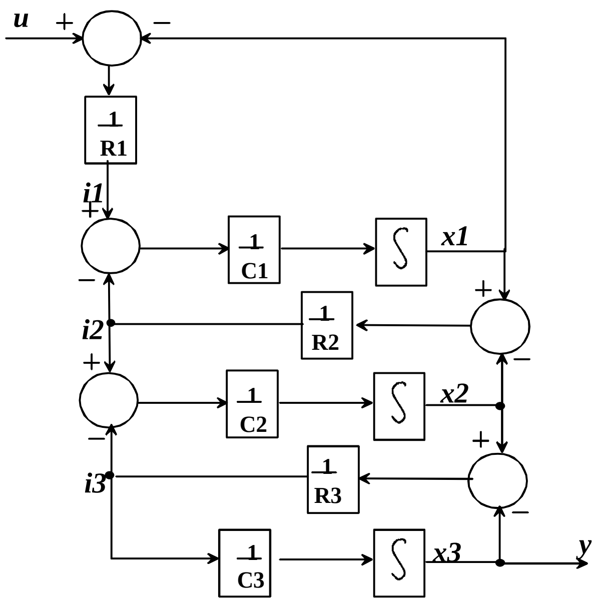


Fig. 13: Block diagram of the RC network

By inspecting the block diagram the network equations can be formulated, and then by using the command solve the (normalized) state space representation is obtained

```
[dx1,dx2,dx3,i1,i2,i3]=solve('i1=(u-x1)/r1', 'i2=(x1-x2)/r2', ...
   'i3=(x2-x3)/r3', 'dx1=(i1-i2)/c1' , 'dx2=(i2-i3)/c2' , ...
   'dx3=i3/c3', 'dx1','dx2','dx3', 'i1','i2','i3')


dx1 =
(-r1*x1+r1*x2+u*r2-x1*r2)/r1/c1/r2

dx2 =
(-r2*x2+r2*x3+x1*r3-x2*r3)/r2/c2/r3

dx3 =
-(-x2+x3)/c3/r3

i1 =
(u-x1)/r1

i2 =
-(-x1+x2)/r2

i3 =
-(-x2+x3)/r3
```

The equations given in the time domain can be easily transformed into the Laplace domain:

```
le(1)=sym('s*x1=dx1');
le(2)=sym('s*x2=dx2');
le(3)=sym('s*x3=dx3');
le(1)=subs(sym(le(1)), 'dx1',dx1);
le(2)=subs(sym(le(2)), 'dx2',dx2);
le(3)=subs(sym(le(3)), 'dx3',dx3);

le(1)
ans =
s*x1 = -(r1*x1-r1*x2-u*r2+x1*r2)/r1/c1/r2

le(2)
ans =
s*x2 = (-r2*x2+r2*x3+x1*r3-x2*r3)/r2/c2/r3

le(3)
ans =
s*x3 = (x2-x3)/c3/r3
```

Finally, solving these equations yields the transfer function

```
[x1,x2,x3]=solve(sym(le(1)),sym(le(2)),sym(le(3)));
y=x3
```

```
y =

u/(s^3*r1*c1*r2*c2*r3*c3+s^2*r1*c1*r2*c2+s^2*r1*c1*r2*c3+s^2*r1*c1*c3*r3+s*r1*c1+r1*
s^2*c2*r3*c3+r1*s*c2+r1*s*c3+s^2*r2*c2*r3*c3+s*r2*c2+r2*s*c3+s*c3*r3+1)
```

```
[num,den]=numden(y);
num=collect(num,'s');
den=collect(den,'s');
pretty(den)
```

```
   3
  s  r1 c1 r2 c2 r3 c3 +

        (r1 c1 r2 c2 + r2 c2 r3 c3 + r1 c2 r3 c3 + r1 c1 r2 c3 + r1 c1 c3 r3)

         2
        s  + (r2 c2 + r1 c2 + r1 c3 + r1 c1 + r2 c3 + c3 r3) s + 1
»
```

6.3 Sampling of continuous systems

A frequent and tedious problem is the conversion of a continuous time transfer function G(s) into the corresponding form of a sampled (discrete time) system G(z). For achieving it the following formula has to be evaluated.

$$G(z) = (1 - \frac{1}{z})\,\mathcal{Z}[\frac{G(s)}{s}]$$

where the operator $\mathcal{Z}$ means the consecutive execution of the following three operators: "Inverse-Laplace-Transformation", "sampling", and "z-Transformation". Of course, this is no problem if you have a lookup table for $\mathcal{Z}$ or a symbolic computation package at hand.

The computation is shown by means of a simple example (see fig. 14).

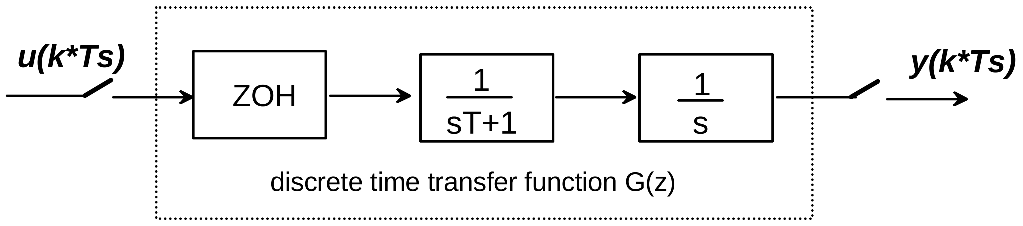


Fig. 14: sampled continuous time system ("zero order hold equivalence")

This script generates the following output and the plot of fig. 15.

```
Gs=sym('1/s^2/(T*s+1)')
gt=ilaplace(Gs)
gt=subs(gt,'t','k*Ts')
Gz1=ztrans(gt)
Gz=symmul(Gz1,'(z-1)/z');
Gz=collect(Gz,'z');
Gz=simplify(Gz)
%
[num,den]=numden(Gz);
% evaluated for T=1.0 and Ts=0.1
num=subs(num, 'T',1.0);
den=subs(den, 'T',1.0);
num=subs(num, 'Ts',0.1);
den=subs(den, 'Ts',0.1);
numN=sym2poly(num)
denN=sym2poly(den)
% use P controller with gain=1
[numcl,dencl]=feedback(numN,denN,1,1);
dstep(numcl,dencl)
xlabel('Time (k*0.1sec)')
ylabel('  ')
```

```
Gs =

1/s^2/(T*s+1)

gt =

-T+t+T*exp(-t/T)

gt =

-T+k*Ts+T*exp(-k*Ts/T)


 Gz1 =

-T*z/(z-1)+Ts*z/(z-1)^2+T*z/exp(-Ts/T)/(z/exp(-Ts/T)-1)


Gz =

-(-T*z+T*z*exp(Ts/T)+T-Ts*z*exp(Ts/T)+Ts-T*exp(Ts/T))/(z*exp(Ts/T)-1)/(z-1)


numN =

    0.0053    0.0052


denN =

    1.1052   -2.1052    1.0000

»
```

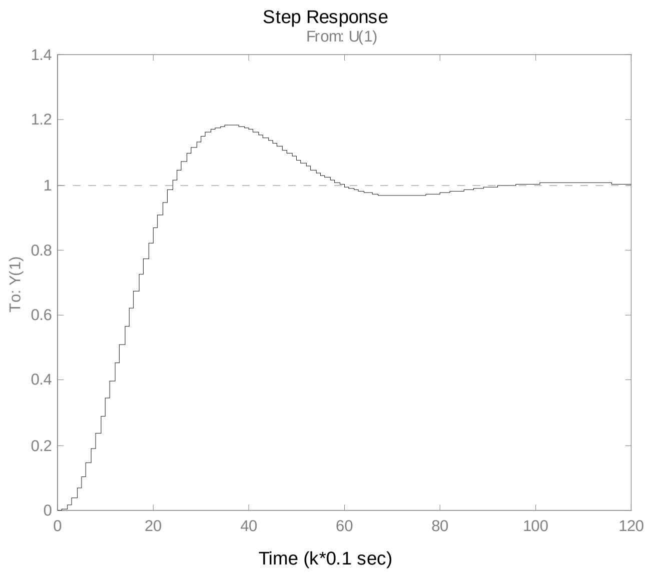


Fig. 15: step response of the closed loop using a P controller