Sammy Omari, Michael Blösch

MATLAB
Control Systems Toolbox
Compendium

October 26, 2007

ETH Zürich

Contents

1

Introduction

1.1 Preface

1.1.1 Objective

This text serves as a compendium for students working with the MATLAB Control Systems Toolbox and Simulink. Its primary objective is to familiarize the students with the toolboxes. Obviously, this can only be achieved by working with MATLAB. That's why this document is complemented with exercises which will be held throughout the semester.

1.1.2 Prerequisites

Students are supposed to have a basic knowledge of MATLAB. This includes vector manipulation, basic programming skills and visualization of data. This text is based on the MATLAB Engineering Toolcourse offered by the IMRT in the third semester for D-MAVT students. The course can be found at www.imrt.ethz.ch/education/matlab.

1.1.3 Structure

The sections are divided in two parts. The function references are followed by examples which discuss the previously introduced functions. The reference is based on the MATLAB help function. Unnecessary details which are not relevant for the students are omitted.

The reference may seem cryptic at first glance, so in order to get a full understanding, consult the examples. In the examples, the implementation of control system concepts and algorithms in MATLAB are discussed. This should provide you with enough information to successfully solve the exercises on your own.

1.2 Tips and Tricks

- When you get stuck writing code, use `help functionname`. If you don't know the exact name, you can use `lookfor` or the graphical help found in the pulldown-menu help.
- Instead of entering the code in the command window, use M-files.
- The first thing to add to an M-file are the commands `clear all, clc, close all`. This will erase all previously defined variables, clears the command window and closes all plot windows.
- Try to use variables instead of numerical values so you can manipulate them more easily later on in the design process.
- Executing an M-file is done by pressing F5. You can abort the computation prematurely using Ctrl-C.
- Comment your source code. It will be a blessing for other people which are reading your code (e.g. your assistants :P).
- All variables can be found in the workspace (the subwindow on the left side of MATLAB). They can be edited easily in an Excel-like application by double-clicking.
- Multiplying two vectors or matrices element-wise is done by the operator `.*` .The point operator can also be used in conjunction with other arithmetic operators, more precisely: `+ - * / \ ^`
- You can save variables in your current directory using `save var`. Loading is done `load var`. Make sure you are in the correct directory.

1.3 Plotting

Although the functions `logspace()` and `plot()` are already introduced in the tools course, they are included here as well, so that the compendium is self-contained. The basic tools for data visualization are reintroduced to refresh the student's skills.

1.3.1 plot()

Purpose

2-D line plot

Syntax

```
plot(Y)
plot(X1,Y1,X2,Y2,...)
plot(X1,Y1,LineSpec,X2,Y2,...)
```

Description

`plot(Y)` plots the columns of `Y` versus their index if `Y` is a real number. If `Y` is complex, `plot(Y)` is equivalent to `plot(real(Y),imag(Y))`. In all other uses of plot, the imaginary component is ignored.

`plot(X1,Y1,X2,Y2,...)` plots all lines defined by `Xn` versus `Yn` pairs.

`plot(X1,Y1,LineSpec,X2,Y2,...)` plots all lines defined before the `LineSpec` argument using the specifications defined in `LineSpec`. This argument gives you control over various graphic characteristics, such as the line style/width, color and marker type/size.

For example, `plot(X1,Y1,'--b')` plots `X1` versus `Y1` using a dashed(`--`), blue (`b`) line. For a complete documentation of `LineSpec`, consult the graphical MATLAB help.

1.3.2 logspace()

Purpose

Generate logarithmically spaced vectors.

Syntax

```
y = logspace(a,b)
y = logspace(a,b,n)
```

Description

The logspace function generates logarithmically spaced vectors. It is especially useful for creating frequency vectors.

`y = logspace(a,b)` generates a row vector `y` of 50 logarithmically spaced points between decades 10^a and 10^b.

`y = logspace(a,b,n)` generates `n` points between the decades 10^a and 10^b.

2

System Definition

2.1 System Representation

The functions `ss()`[1], `tf()`[2], `zpk()`[3], and `frd()`[4] create transfer function models, zero-pole-gain models, state-space models, and frequency response data models, respectively. These functions take the model data as input and produce TF, ZPK, SS, or FRD objects that store this data in a single MATLAB variable. This section shows how to create continuous SISO or MIMO LTI models using `ss()`, `tf()`, `zpk()`, and `frd()`.

2.1.1 ss()

Purpose

Specifies state-space models or converts LTI model to state space.

Syntax

```
sys_ss = ss(A,B,C,D)
sys_ss = ss(sys_tf)
sys_ss = ss(sys_zpk)
```

[1] see 2.1.1 ss()
[2] see 2.1.2 tf()
[3] see 2.1.3 zpk()
[4] see 2.1.4 frd()

Description

Returns a real- or complex-valued state-space model (SS object) or converts a transfer function or a zero-pole-gain model to state space.

Creates a continuous-time state-space model

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

For a model with n states, p outputs, and m inputs, the input matrices A,B,C,D have to be:

- $A = \mathbb{R}^{n \times n}$
- $B = \mathbb{R}^{n \times m}$
- $C = \mathbb{R}^{p \times n}$
- $D = \mathbb{R}^{p \times m}$

The input can also be a transfer function (TF object) or a zero-pole-gain model (ZPK object). In this case the system matrices `A`,`B`,`C`,`D` are derived from the transfer function.

2.1.2 tf()

Purpose

Creates or converts to transfer function model.

Syntax

```
sys_tf = tf(num,den)
sys_tf = tf(sys_ss)
sys_tf = tf(sys_zpk)
s = tf('s')
```

Description

Returns a real- or complex-valued transfer function model (TF object) or converts a state-space or a zero-pole-gain model to transfer function form.

Creates a continuous-time transfer function with numerator and denominator specified by `num` and `den`.

In the SISO case, num and den are the real- or complex-valued row vectors of numerator and denominator coefficients ordered in descending powers of `s`.

$$h(s) = \frac{b_m s^m + b_{m-1} s^{m-1} + \ldots + b_2 s^2 + b_1 s + b_0}{a_n s^n + a_{n-1} s^{n-1} + \ldots + a_2 s^2 + a_1 s + b_0}$$

$$num = [b_m, b_{m-1}, \ldots, b_2, b_1, b_0]$$
$$den = [a_n, a_{n-1}, \ldots, a_2, a_1, a_0]$$

In a MIMO system `num` and `den` are cell arrays of row vectors with as many rows as outputs and as many columns as inputs. Each `num(i,j)`/`den(i,j)` pair specifies the transfer function from input `j` to output `i`.

You can also use real- or complex-valued rational expressions to create a transfer function model. Type `s = tf('s')`, before specifing a transfer function model using a rational function in the Laplace variable, `s`.

The input can also be a state-space model (SS object) or a zero-pole-gain model (ZPK object).

2.1.3 zpk()

Purpose

Creates or converts to zero-pole-gain model.

Syntax

```
sys_zpk = zpk(z,p,k)
sys_zpk = zpk(sys_ss)
sys_zpk = zpk(sys_tf)
s = zpk('s')
```

Description

Returns a real- or complex-valued zero-pole-gain model (ZPK object) or converts a state-space or a transfer function model to zero-pole-gain form.

Creates a continuous-time zero-pole-gain model with zeros `z`, poles `p`, and gain `k`.

In the SISO case, `z` and `p` are the vectors of real- or complex-valued zeros and poles, and `k` is the real- or complex-valued scalar gain.

$$h(s) = k\frac{(s - z_1)(s - z_2)\ldots(s - z_{m-1})(s - z_m)}{(s - p_1)(s - p_2)\ldots(s - p_{n-1})(s - p_n)}$$

$$z = [z_1, z_2, \ldots, z_{m-1}, z_m]$$
$$p = [p_1, p_2, \ldots, p_{n-1}, p_n]$$

In a MIMO system `z` and `p` are cell arrays of vectors with as many rows as outputs and as many columns as inputs, and `k` is a matrix with as many rows as outputs and as many columns as inputs. The vectors `z(i,j)` and `p(i,j)`

specify the zeros and poles of the transfer function from input `j` to output `i`. The command `k(i,j)` specifies the (scalar) gain of the transfer function from input `j` to output `i`.

You can also use real- or complex-valued rational expressions to create a zero-pole-gain model. Type `s = zpk('s')`, before specifing a zero-pole-gain model using a rational function in the Laplace variable, `s`.

The input can also be a state-space model (SS object) or a transfer function (TF object).

2.1.4 frd()

Purpose

Creates or converts to frequency-response data model.

Syntax

```
sys_frd = frd(response,frequency)
sys_frd = frd(sys_ss,frequency)
sys_frd = frd(sys_tf,frequency)
sys_frd = frd(sys_zpk,frequency)
```

Description

The command `sys = frd(response,frequency)` creates an frequency-response data model from the frequency response data stored in the multidimensional array
`response`. The vector `frequency` represents the underlying frequencies for the frequency response data.

In a SISO model, `response` is a vector of length l for which `response(i)` is the frequency response at the frequency `frequency(i)`.

In MIMO, `response` is a p-by-m-by-l multidimensional array (remember: $D = \mathbb{R}^{p\times m}$) for which `response(i,j,k)` specifies the frequency response from input `j` to output `i` at frequency `frequency(k)`.

The command `sysfrd = frd(sys,frequency)` converts a transfer function, state-space, or zero-pole-gain model to a frequency-response data model. The frequency response is computed at the frequencies provided by the vector `frequency`.

2.1.5 Example: Inverted Pendulum on a Cart [5]

The inverted pendulum on a cart is a typical SISO problem for a control systems engineer. It can easily be shown that the problem is highly unstable.

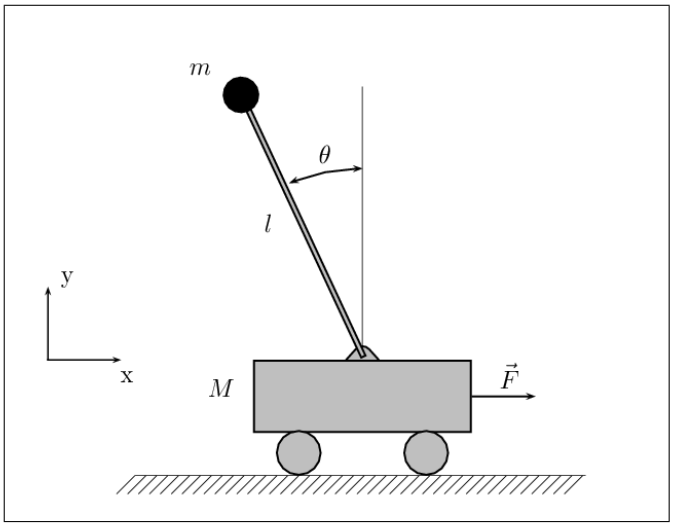


Fig. 2.1. Pendulum on a Cart

This exercise can be seen as a simplification of a lot of real world applications as for example the Segway or a starting rocket.

Both the system matrices A, B, C, D and the derived transfer function $P(s)$ are given:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & -\frac{gm}{M} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & \frac{g(M+m)}{lM} & 0 \end{pmatrix}, B = \begin{pmatrix} 0 \\ -\frac{1}{M} \\ 0 \\ -\frac{1}{lM} \end{pmatrix}, C = \begin{pmatrix} 0 & 0 & 1 & 0 \end{pmatrix}, D = 0$$

$$P(s) = \frac{-1}{Mls^2 - g(m+M)} = -\frac{1}{Ml} \frac{1}{(s + \sqrt{\frac{g(M+m)}{Ml}})(s - \sqrt{\frac{g(M+m)}{Ml}})}$$

Assume the pendulum's length is $l = 0.5m$, the mass at the end is $m = 2kg$ and the cart weighs $M = 4kg$. Therefore, we write an M-File to save the matrices in a MATLAB object for further manipulation.

In a second step, we assume that we don't know anything about the state variables. Only the input-output behaviour represented either by its zeros/poles or its polynomial coefficients is known.

M-file

```
clear all, clc

% Parameters:
```

[5] see chapter 4.4.3, Analysis and Synthesis of SISO Control Systems

```
M=4;                            % Weight of the cart [kg]
m=2;                            % Weight of the upper mass [kg]
l= 0.5;                         % Length of the pendulum [m]
g=10;                           % Gravitational constant [m/s^2]

% State-space matrices:
A=[0 1 0 0;0 0 g*m/M 0;0 0 0 1;0 0 -g*(m+M)/(l*M) 0];
B=[0;1/M;0;1/l*M];
C=[0 0 1 0];
D=0;

% Build state-space object:
sys_ss = ss(A,B,C,D);

% Transform to tf model:
sys_tf_ss=tf(sys_ss)

% Transform to zpk model:
sys_zpk_ss=zpk(sys_ss);

% Create tf model:
s=tf('s');
sys_tf= -1/((M*l*s^2 -g*(m+M)));

% Create zpk model:
z = [];                                          % Zeros
p = [sqrt(g*(M+m)/(M*l)) -sqrt(g*(M+m)/(M*l))];  % Poles
k= -1/(l*M);                                     % Gain
sys_zpk= zpk(z,p,k)

% Compute controller canonical form:
sys_ss_tf_ss = ss(sys_tf_ss);
A_ss_tf_ss = sys_ss_tf_ss.A
```

Output:

```
Transfer function:

-0.5

--------

s^2 - 30


Zero/pole/gain:

-0.5

-------------------

(s-5.477) (s+5.477)


A_ss_tf_ss =
     0  7.5000
4.0000       0
```

Explanation

In the M-File, we define all parameters, such as the weight and the length of the pendulum. As engineers, we often have to change those parameters, so it would be unwise to just enter the numerical matrices.

With the given state-space matrices, we produce an SS model using the function `ss()`[6]. The resulting SS object is further transformed into TF and ZPK models using the command `tf()`[7] or `zpk()`[8], respectively.

The LTI model can also be computed using the polynomial coefficients of the transfer function or its zeros, poles and gain. The command `s=tf('s')` simplifies building new TF models. Afterwards, we can simply enter the transfer function as a rational fraction.

Having the zeros, poles and the gain of the transfer function, we can easily obtain a ZPK model using the command `zpk(z,p,k)`, where `z`, `p` and `k` are vectors where the zeros, poles and gain are stored.

The different LTI models can all be transformed into each other (except the FRD LTI model). But we have to be careful when transforming a TF or ZPK model into an SS model. The transformation of an SS model to a TF model and back in an SS model again (`ss(sys_tf_ss)`) can result in a pole-zero cancellation and therefore a loss of information about the internal states of the system. As we can see, the realized dynamic matrix (`ss(sys_tf_ss).A`) has only dimension 2, whereas the original matrix had dimension 4.

2.1.6 Example: Deriving a nominal model using frd()

Instead of modeling a system, we now measure the steady-state response at a given frequency range.

We are using a sinusoidal input u with a frequency ω and measure the amplitude and the phase of the outptut y h times for each ω. For further manipulation in MATLAB, we store the measured data in the file `data.mat` where we can access the matrix `mag`, `phase` and `w`. The resulting matrices `mag` and `phase` have the dimension $\mathbb{R}^{l \times h}$ where l is the number of frequencies and h is the number of the series of measurements. The vector `w` has therefore dimension $\mathbb{R}^{l}$. With the resulting data, we try to derive a transfer function of the nominal model.

[6] see 2.1.1 ss()
[7] see 2.1.2 tf()
[8] see 2.1.3 zpk()

M-file

```
clear all, clc, close all

% Import matrices mag,phase,w:
load data

% Transform measured data into FRD model:
z=mag.*exp(i*phase);                          % Vector with complex form of the response
sys_frd = frd(z,w);

% Plot measured data:
figure(1)
bode(sys_frd,'.r'),hold on

% Derive nominal model that fits data:
s=tf('s');
sys_nom=20/(s^2+2*s+20);

% Plot nominal model:
bode(sys_nom,'k'), hold on
```

Output:

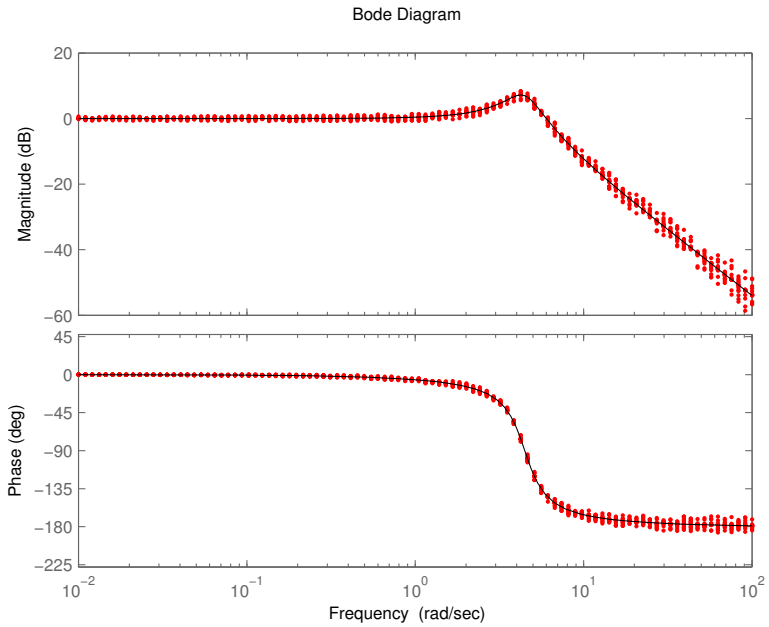


Fig. 2.2. Bode plot of data (dots) and nominal system (solid line)

Explanation:

The commands `clear all, clc, close all` remove all stored variables in MATLAB, remove the text in the command window and close all open figures. It should be included in every M-file to minimize any complications with previously executed M-files.

After importing the data from the file `data.mat` using `load data`, we transform it in an FRD model `sys_frd`. We therefore compute the complex form of the response using the amplitude and phase information. Mathematically spoken:

$$z = r\angle\varphi = re^{i\varphi}, r = magnitude, \varphi = phase$$

The way it is done in MATLAB is to compute every complex response using the corresponding phase and magnitude elements, stored in the matrices `phase` or `mag`, respectively. We use the MATLAB operator `.*` for element-wise multiplication. We can now create the FRD object `sys_frd` using the function `frd(z,w)`[9] where `z` is the complex response matrix and `w` the frequency vector.

With `bode(sys_frd,'.r')`[10], we plot the Bode diagram of `sys_frd` using red dots. The output is depicted in FIg. 2.2. As you can see, the model resembles a second-order system. After a few iterations, a nominal system `sys_nom` is found that fits the measured data. With the command `hold on`, we can display both Bode plots in the same figure.

2.2 System Interconnections

The Matlab control system toolbox offers various functions to connect LTI models. In the following section we introduce the three functions `series()`[11], `parallel()`[12] and `feedback()`[13].These can be very useful for creating larger systems based on smaller subsystems, for example a plant with its controller.

2.2.1 series()

Purpose

Connects two LTI models in series.

[9] see 2.1.4 frd()
[10] see 3.3.4 bode()
[11] see 2.2.1 series()
[12] see 2.2.2 parallel()
[13] see 2.2.3 feedback()

Syntax

```
sys = series(sys1,sys2)
sys = series(sys1,sys2,out1,inp2)
```

Description

`series` connects two LTI models in series. This function accepts any type of LTI model.

`sys = series(sys1,sys2)` forms the basic series connection. This command is equivalent to the direct multiplication `sys = sys2 * sys1`.

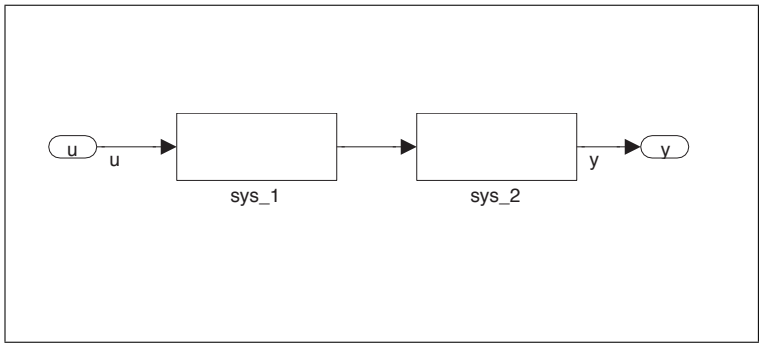


Fig. 2.3. Two LTI models in series

`sys = series(sys1,sys2,out1,inp2)` forms the more general series connection. The index vectors `out1` and `inp2` indicate which outputs `y_1` of `sys1` and which inputs `u_2` of `sys2` should be connected. The resulting model `sys` has `[u ; v_2]` as input and `[z_1 ; y]` as output.

2.2.2 parallel()

Purpose

Connects two LTI models in parallel.

Syntax

```
sys = parallel(sys1,sys2)
sys = parallel(sys1,sys2,inp1,inp2,out1,out2)
```

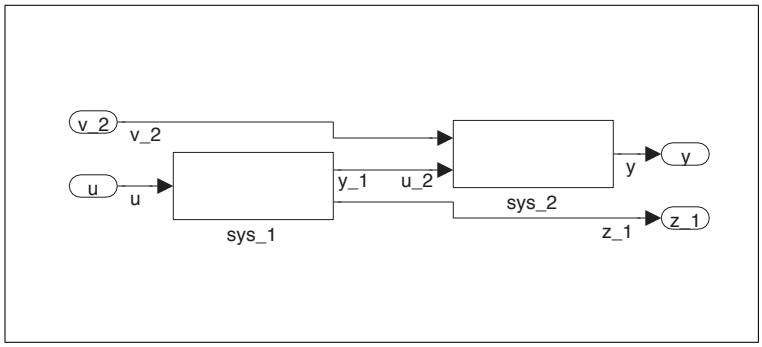


Fig. 2.4. General form of two LTI models in series

Description

`parallel` connects two LTI models in parallel. This function accepts any type of LTI model.

`sys = parallel(sys1,sys2)` forms the basic parallel connection. This command is equivalent to the direct addition `sys = sys1 + sys2`.

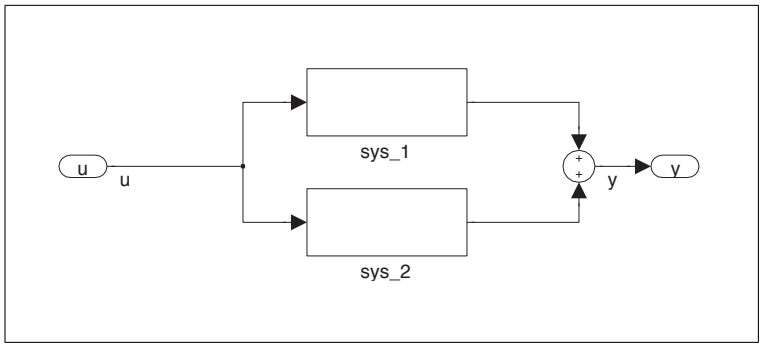


Fig. 2.5. Two LTI models in parallel

`sys = parallel(sys1,sys2,inp1,inp2,out1,out2)` forms the more general parallel connection. The index vectors `inp1` and `inp2` specify which inputs `u_1` of `sys1` and which inputs `u_2` of `sys2` are connected. Similarly, the index vectors `out1` and `out2` specify which outputs `y_1` of `sys1` and which outputs `y_2` of `sys2` are summed. The resulting model `sys` has `[v_1 ; u ; v_2]` as inputs and `[z_1 ; y ; z_2]` as outputs.

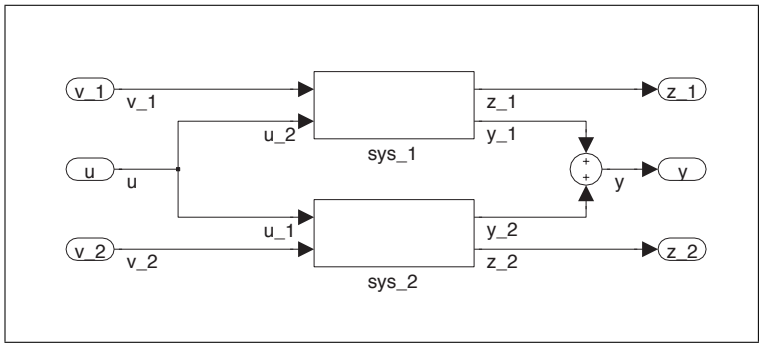


Fig. 2.6. General form of two LTI models in parallel

2.2.3 feedback()

Purpose

Applies a feedback interconnection on two LTI models.

Syntax

```
sys = feedback(sys1,sys2)
sys = feedback(sys1,sys2,feedin,feedout)
```

Description

`sys = feedback(sys1,sys2)` returns an LTI model `sys` for the negative feedback interconnection. The closed-loop model `sys` has `u` as input vector and `y` as output vector.

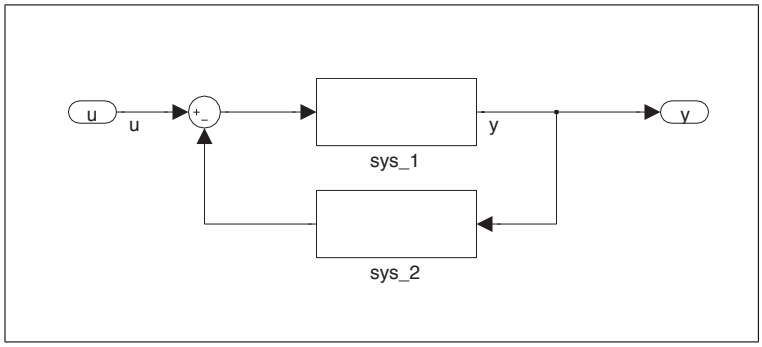


Fig. 2.7. Feedback connection of two LTI models

`sys = feedback(sys1,sys2,feedin,feedout)` computes a closed-loop model `sys` for the more general feedback loop. The vector `feedin` contains indices into the input vector of `sys1` and specifies which inputs `u` are involved in the feedback loop. Similarly, `feedout` specifies which outputs `y` of `sys1` are used for feedback. The resulting LTI model `sys` has the same inputs and outputs as `sys1` (with their order preserved).

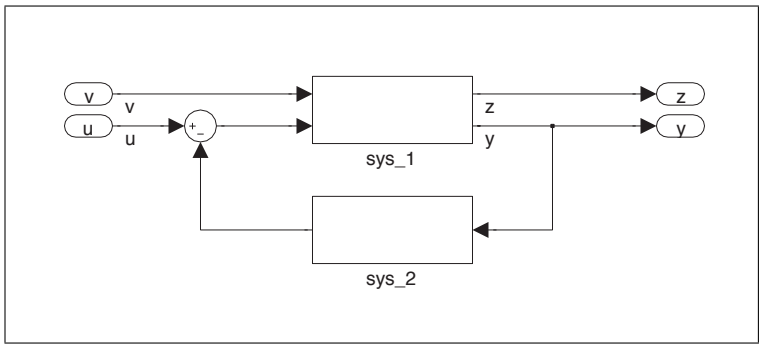


Fig. 2.8. General form of feedback connection between two LTI models

2.2.4 Example: Connecting SISO systems

We define three arbitrary systems `sys_1`, `sys_2` and `sys_3` which we will connect in the order as shown in Fig. 2.9. In a last step we compute the closed-loop behaviour using `feedback()`[14].

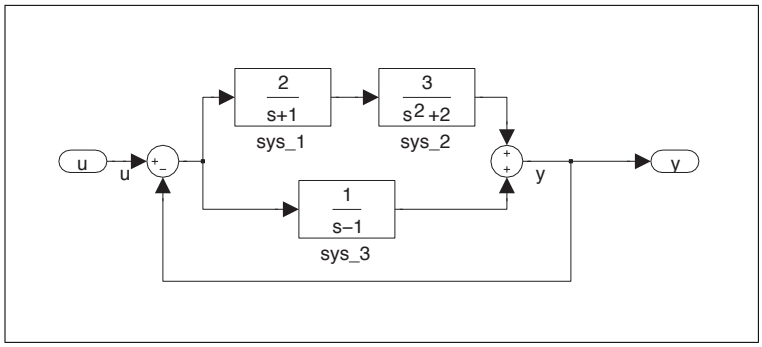


Fig. 2.9. Connecting three SISO systems

M-file

```
clear all, clc

% Define systems:
s=tf('s');
sys_1=2/(s+1);
sys_2=3/(s^2+2);
sys_3=1/(s-1);
sys_12=series(sys_1,sys_2)                % Series connection of sys_1 and sys_2
sys_123=parallel(sys_12,sys_3)            % Parallel connection of sys_12 and sys_3
sys_fb=feedback(sys_123,1)                % Feedback of sys_123
```

Output:

```
Transfer function:
6
-------------------
s^3 + s^2 + 2 s + 2


Transfer function:
s^3 + s^2 + 8 s - 4
-------------------
s^4 + s^2 - 2
```

[14] see 2.2.3 feedback()

```
Transfer function:
s^3 + s^2 + 8 s - 4
---------------------------
s^4 + s^3 + 2 s^2 + 8 s - 6
```

Explanation:

As usual, we define `s` as a transfer function to simplify building the systems. Next, we compute the transfer function of the upper branch using `sys_12 = series(sys_1,sys_2)`[15] which connects the systems `sys_1` and `sys_2` in series. The resulting system is used in the subsequent command `parallel(sys_12,sys_3)`[16] to get the open-loop input-output behaviour of the complete system `sys_123`. To get the closed-loop transfer function `sys_fb`, we use the function `feedback(sys_123,1)`[17]which feeds the output signal back to its input. Please note that the output is substracted from the input, which is already done by the function `feedback()` (hence the `1` as argument).

[15] see 2.2.1 series()
[16] see 2.2.2 parallel()
[17] see 2.2.3 feedback()

3

System Analysis and Control Design

3.1 Controllability and Observability

The MATLAB Control System Toolbox provides the functions `obsv()`[1] and `ctrb()`[2] to relieve the user from the painstaking task of manually computing the observability and controllability matrices.

3.1.1 obsv()

Purpose

Computes the observability matrix.

Syntax

```
ob = obsv(A,C)
ob = obsv(sys)
```

Description

Returns the observability matrix `ob` for a state space system. For an n-by-n matrix `A` and a p-by-n matrix `C`, `obsv(A,C)` returns the observability matrix

$$\mathcal{O} = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

[1] see 3.1.1 obsv()
[2] see 3.1.2 ctrb()

The resulting observability matrix `ob` has n columns and $n \cdot p$ rows and can therefore also be used for MIMO systems (which obviously have $p > 1$).

The model is observable if the matrix `ob` has full rank n.

`obsv(A,C)` is equivalent to `obsv(sys)` where `sys = ss(A,B,C,D)`[3].

3.1.2 ctrb()

Purpose

Computes the Controllability matrix.

Syntax

```
co = ctrb(A,B)
co = ctrb(sys)
```

Description

Returns the controllability matrix `co` for a state space system. For an n-by-n matrix `A` and a n-by-m matrix `B`, `ctrb(A,B)` returns the controllability matrix

$$\mathcal{R} = \begin{bmatrix} B & AB & A^2B & \cdots & A^{n-1}B \end{bmatrix}$$

The resulting controllability matrix `co` has n rows and $n \cdot m$ columns and can therefore also be used for MIMO systems (which obviously have $m > 1$).

The model is controllable if the matrix `co` has full rank n.

`ctrb(A,B)` is equivalent to `ctrb(sys)` where `sys = ss(A,B,C,D)`[4].

3.1.3 rank()

Purpose

Computes the rank of the matrix.

Syntax

```
k = rank(A)
```

Description

Returns the number of linearly independent rows or columns `k` of a full matrix `A`.

[3] see 2.1.1 ss()
[4] see 2.1.1 ss()

3.1.4 SISO Example: Inverted Pendulum on a Cart [5]

We use the example of the inverted pendulum to show that it is highly important to check whether a system is observable and controllable. If an unstable state is not controllable, there is no possibility to stabilize a system. Therefore, we write an M-File to check if the observability and controllability matrices have full rank.

M-file

```
clear all, clc

% Parameters:
M=4;                            % Weight of the cart [kg]
m=2;                            % Weight of the upper mass [kg]
l= 0.5;                         % Length of the pendulum [m]
g=10;                           % Gravitational constant [m/s^2]

% State-space matrices:
A=[0 1 0 0;0 0 g*m/M 0;0 0 0 1;0 0 -g*(m+M)/(l*M) 0];
B=[0;1/M;0;1/l*M];
C=[0 0 1 0];
D=0;

% Observability:
ob = obsv(A,C);
r_ob=rank(ob)

% Controllability:
co = ctrb(A,B);
r_co = rank(co)
```

Output:

```
r_ob = 2

r_co = 4
```

Explanation

With the given state-space matrices, the observability and controllability matrices are computed. To check if the system can both be controlled and observed, we use the MATLAB functions `obsv()`[6] and `ctrb()`[7]. Using the function `rank()`[8] we realize that the observability matrix doesn't have full rank (dimension of `ob` is 4, the rank only 2!).

[5] see chapter 4.4.3, Analysis and Synthesis of SISO Control Systems
[6] see 3.1.1 obsv()
[7] see 3.1.2 ctrb()
[8] see 3.1.3 rank()

Instead of measuring the angle of the pendulum, we use another sensor to measure the position of the cart. Therefore, we have a new matrix `C_new`.

$$y_{new} = x_1,\ C_{new} = \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix}$$

Another computation with `C_new` shows that now the observability matrix has full rank.

3.2 System Properties

To design a controller for a dynamic system, its properties have to be known. The toolbox offers various functions to support the control systems engineer. The commands `eig()`[9] and `svd()`[10] can be used to analyze the dynamic system matrix A, whereas the functions `sigma()`[11], `pole()`[12], `zero()`[13], `pzmap()`[14] and `evalfr()`[15] analyse the frequency domain properties of the transfer functions.

3.2.1 eig()

Purpose

Finds eigenvalues and eigenvectors.

Syntax

```
d = eig(A)
[V,D] = eig(A)
```

Description

`d = eig(A)` returns a vector of the eigenvalues of matrix `A`.

`[V,D] = eig(A)` produces matrices of eigenvalues `D` and eigenvectors `V` of matrix `A`, so that $A \cdot V = V \cdot D$. Matrix `D` is the canonical form of `A` — a diagonal matrix with `A`'s eigenvalues on the main diagonal. Matrix `V` is the modal matrix — its columns are the eigenvectors of `A`.

[9] see 3.2.1 eig()
[10] see 3.2.2 svd()
[11] see 3.2.3 sigma()
[12] see 3.2.4 pole()
[13] see 3.2.5 zero()
[14] see 3.2.6 pzmap()
[15] see 3.2.7 evalfr()

3.2.2 svd()

Purpose

Computes the matrix singular value decomposition.

Syntax

```
s = svd(X)
[U,S,V] = svd(X)
```

Description

`s = svd(X)` returns a vector of singular values of `X`.

`[U,S,V] = svd(X)` produces a diagonal matrix `S` of the same dimension as `X`, with nonnegative diagonal elements in decreasing order, and unitary matrices `U` and `V` so that $X = U \cdot S \cdot V^T$.

3.2.3 sigma()

Purpose

Plot singular values of LTI models.

Syntax

```
sigma(sys)
sigma(sys,w)
```

Description

`sigma` calculates the singular values of the frequency response of an LTI model. For an FRD model `sys`, `sigma` computes the singular values of the response of `sys`. For continuous-time TF, SS, or ZPK models with transfer function `H(s)`, `sigma` computes the singular values of `H(jw)` as a function of the frequency `w`.

`sigma(sys)` plots the singular values of the frequency response of an arbitrary LTI model `sys`.

`sigma(sys,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval `[wmin,wmax]`, set `w = {wmin,wmax}`. To use particular frequency points, set `w` to the corresponding vector of frequencies. Use `logspace`[16] to generate logarithmically spaced frequency vectors. The frequencies must be specified in rad/s.

[16] see 1.3.2 logspace()

3.2.4 pole()

Purpose

Computes poles of LTI system.

Syntax

```
p = pole(sys)
```

Description

`pole` computes the poles `p` of the SISO or MIMO LTI model `sys`.

For state-space models, the poles are the eigenvalues of the `A` matrix, or the generalized eigenvalues of $A - \lambda E$ in the descriptor case.

For SISO transfer functions or zero-pole-gain models, the poles are simply the denominator roots.

For MIMO transfer functions (or zero-pole-gain models), the poles are computed as the union of the poles for each SISO entry. If some columns or rows have a common denominator, the roots of this denominator are counted only once.

3.2.5 zero()

Purpose

Computes zeros of LTI system.

Syntax

```
z = zero(sys)
```

Description

`zero` computes the zeros of SISO systems and the transmission zeros of MIMO systems. For a MIMO system with matrices `(A,B,C,D)`, the transmission zeros are the complex values λ for which the normal rank of

$$\begin{bmatrix} A - \lambda E & B \\ C & D \end{bmatrix}$$

drops.

`z = zero(sys)` returns the (transmission) zeros of the LTI model `sys` as a column vector.

3.2.6 pzmap()

Purpose

Compute pole-zero map of LTI models.

Syntax

```
pzmap(sys)
[p,z] = pzmap(sys)
```

Description

`pzmap(sys)` plots the pole-zero map of the continuous- or discrete-time LTI model `sys`. For SISO systems, `pzmap` plots the transfer function poles and zeros. For MIMO systems, it plots the system poles and transmission zeros. The poles are plotted as x's and the zeros are plotted as o's.

`[p,z] = pzmap(sys)` returns the system poles and (transmission) zeros in the column vectors `p` and `z`.

3.2.7 evalfr()

Purpose

Evaluates the frequency response at any given frequency.

Syntax

```
frsp = evalfr(sys,z)
```

Description

`frsp = evalfr(sys,z)` evaluates the transfer function of the TF, SS, or ZPK model `sys` at the frequency `z`. Please note that the frequency is always a complex number.

3.2.8 Example: Levitating Sphere [17]

In this example, we want to study the system properties of a levitating sphere. Such a sphere basically consists of a ferromagnetic material and floats below a strong electromagnet. The gravitational and the magnetic force are acting on the sphere. To control the sphere, we apply a voltage to the electromagnet.

[17] see chapter 3.10, Analysis and Synthesis of MIMO Control Systems

This results in a force on the sphere. We therefore choose the voltage as input $u(t)$ and the deviation of the sphere from its nominal position as output $y(t)$.

After modeling the system, we get the following system matrices:

$$A = \begin{pmatrix} 0 & 1 & 0 \\ 700 & 0 & 700 \\ 0 & 0 & -0.2 \end{pmatrix}, B = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, C = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}, D = 0$$

The internal states are the position ($x1$) and the velocity ($x2$) of the sphere and the current ($x3$) in the electromagnet.

M-File

```
clear all, clc

% State-space matrices:
A=[0 1 0;700 0 700;0 0 -0.2];
B=[0 0 1]';
C=[1 0 0];

% Eigenvalues:
[V,D]=eig(A);
d=diag(D)           % Get the eigenvalues from the diagonal matrix D and store them in d

% Save as a state-space object:
sys_ss = ss(A,B,C,0);

% Poles
p=pole(sys_ss)

% Zeros
z=zero(sys_ss)

% Mapping the Poles and Zeros
pzmap(sys_ss)
```

Output:

```
d =
  26.4575
 -26.4575
  -0.2000
p =
  26.4575
 -26.4575
  -0.2000
z =
Empty matrix: 0-by-1
```

Explanation

After entering the system matrices, we compute the eigenvectors and their corresponding eigenvalues. The eigenvalues are stored on the diagonal of the matrix `D`. We extract them in a vector using the function `diag()`. Next, we

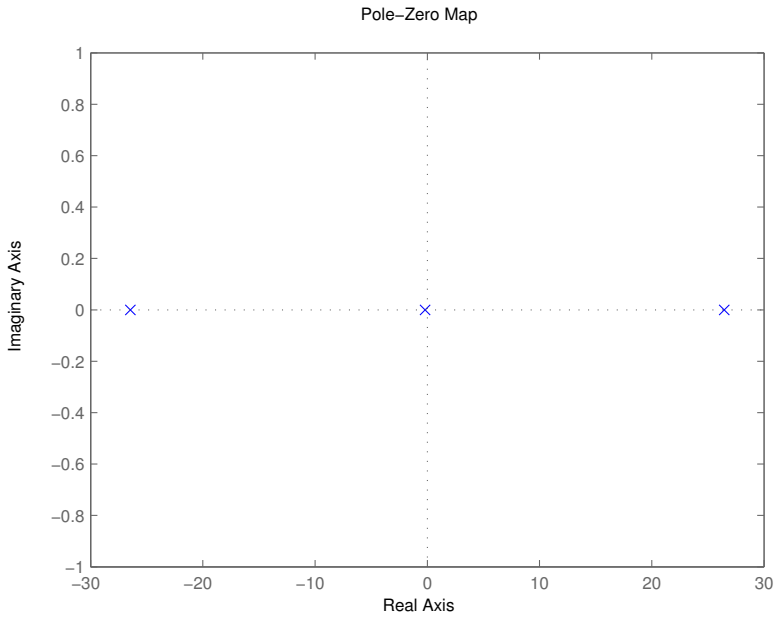


Fig. 3.1. Pole-Zero map of the levitating sphere

directly compute poles of the transfer function. Please note that the poles of the transfer function equal the eigenvalues of the dynamic matrix. This is only the case when no pole-zero cancellation occurs.

As a next step, we compute the zeros of `sys_ss`. The somehow cryptic result tells us that the transfer function has no zeros. This is also confirmed by the pole-zero map which was generated using the function `pzmap(sys_ss)`[18]. Fig. 3.1 us nicely that all poles lie on the real axis.

3.2.9 Example: Geostationary Satellite [19]

If we want to watch satellite TV, we have to install a satellite dish which points directly to the satellite. The reason why we never have to change the alignment of our dish lies in the fact that the satellites are placed in a geostationary orbit. Meaning: The satellite never changes its relative position to the earth. The satellites are all placed on a circular orbit at $r_0 \approx 4.22 \cdot 10^7\, m$ distance from the center of the earth and rotate with the sidereal angular velocity $\omega_0 \approx 7.29 \cdot 10^{-5} rad/s$ around the earth. Because we have to regulate both the radial and the angular velocity using two thrusters, it is a MIMO system.

[18] see 3.2.6 pzmap()
[19] see chapter 3.11, Analysis and Synthesis of MIMO Control Systems

The system dynamics are defined by:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 3\omega_0^2 & 0 & 0 & 2r_0 \cdot \omega_0 \\ 0 & 0 & 0 & 1 \\ 0 & -2\omega_0/r_o & 0 & 0 \end{pmatrix}, B = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1/r_0 \end{pmatrix}, C = \begin{pmatrix} 1/r_0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, D = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

M-File

```
clear all, clc

% Parameters:
r0=4.22*10^7;                   % Set value for the orbit radius [m]
w0=7.29*10^-5;                  % Set value for the angular velocity radius [rad/s]

% System Matrices
A=[0 1 0 0;3*w0^2 0 0 2*r0*w0;0 0 0 1;0 -2*w0/r0 0 0];
B=[0 0;1 0;0 0;0 1/r0];
C=[1/r0 0 0 0;0 0 1 0];
D=zeros(2);

% System Representation:
sys_ss=ss(A,B,C,D);

% Eigenvalues:
d=eig(A)

% Singular values:
M=evalfr(sys_ss,10);            % MIMO transfer function at 10 rad/s
s1=svd(M)
s2=sigma(sys_ss,10)
sigma(sys_ss)                   % Plot the singular values of the system
```

Output:

```
d =

1.0e-004 *

0

0 + 0.7290i

0 - 0.7290i

0

s1 =

1.0e-009 *

0.2370

0.2370

s2 =

1.0e-009 *

0.2370

0.2370
```

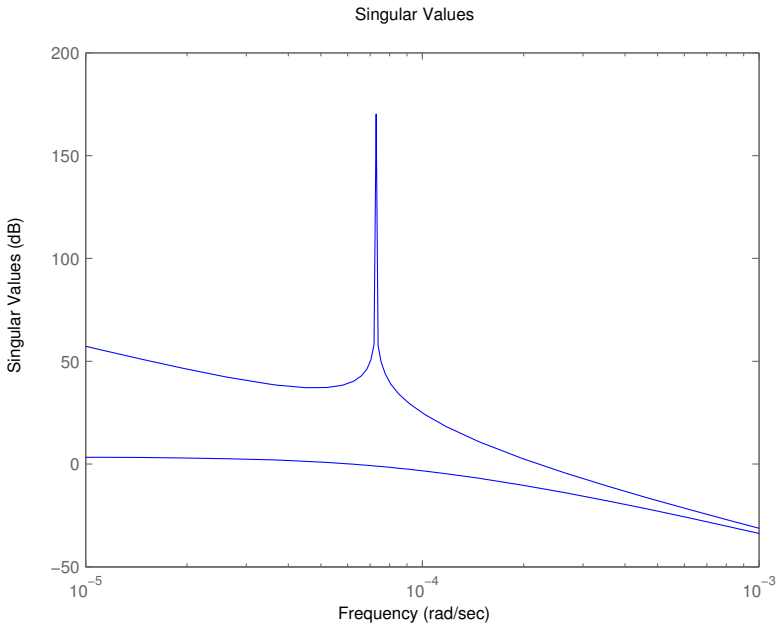


Fig. 3.2. Singular values of the satellite

Explanation

After entering the system matrices and their parameters, we calculate the eigenvalues using `eig(A)`[20]. Because two of the eigenvalues are zero, the system is unstable. [21]

Next, we want to calculate the maximal and minimal singular values at the (arbitrary) frequency $w = 10\ rad/s$. This is done using the function `evalfr(sys_ss,10)`[22] to evaluate the transfer function `sys_ss` at the frequency `10` rad/s. In our (MIMO) case, the resulting matrix `M` has dimension 2. In the next step, we compute the singular values using `svd(M)`[23] to check in which interval the amplitude of the output will be for $w = 10\ rad/s$. As you can see, the singular values are the same, which points to the conclusion that the system has a SISO system behaviour at that frequency.

To simplify the computation of singular values, we use the function `sigma(sys_ss,10)`[24], which returns the same output. We use the same func-

[20] see 3.2.1 eig()
[21] see section 3.3.9 for a controller design
[22] see 3.2.7 evalfr()
[23] see 3.2.2 svd()
[24] see 3.2.3 sigma()

tion in the next step to plot the singular values over a frequency interval (Fig. 3.2).

3.3 System Response

The response of a system can be analysed in the time- and frequency-domains to study the reaction to a previously defined input. The function for a time-domain analysis are `step()`[25], `impulse()`[26] and `initial()`[27]. The frequency-domain responses can be interpreted using `bode()`[28], `margin()`[29], `dcgain()`[30] and `nyquist()`[31].

3.3.1 step()

Purpose

Plots the step response of LTI systems.

Syntax

```
step(sys)
step(sys,t)
step(sys1,sys2,...,sysN)
[y,t] = step(sys)
```

Description

`step(sys)` plots the step response of an arbitrary LTI model `sys`. This model can be SISO or MIMO. The step response of multi-input systems is the collection of step responses for each input channel. The duration of the simulation is determined automatically based on the system poles and zeros.

`step(sys,t)` sets the simulation horizon explicitly. You can specify either a final time `t = Tfinal` (in seconds), or a vector of evenly spaced time samples of the form `t = 0:dt:Tfinal`.

To plot the step responses of several LTI models `sys1,..., sysN` on a single figure, use

```
step(sys1,sys2,...,sysN)
step(sys1,sys2,...,sysN,t)
```

[25] see 3.3.1 step()
[26] see 3.3.2 impulse()
[27] see 3.3.3 initial()
[28] see 3.3.4 bode()
[29] see 3.3.5 margin()
[30] see 3.3.6 dcgain()
[31] see 3.3.7 nyquist()

When invoked with output arguments, the functions

```
[y,t] = step(sys)
[y,t,x] = step(sys)
y = step(sys,t)
```

return the output response `y`, the time vector `t` used for simulation, and the state trajectories x. No plot is drawn on the screen. For single-input systems, `y` has as many rows as time samples (`length(t)`) and as many columns as outputs. In the multi-input case, the step responses of each input channel are stacked up along the third dimension of `y`. The dimensions of `y` are then

$$(length\ of\ t) \times p \times m$$

and `y(:,:,j)` gives the response to a unit step command injected in the `j`th input channel. Similarly, the dimensions of `x` are

$$(length\ of\ t) \times n \times m$$

where n is the number of states, m the number of inputs and p the number of outputs.

3.3.2 impulse()

Purpose

Plots the impulse response of LTI systems.

Syntax

```
impulse(sys)
impulse(sys,t) impulse(sys1,sys2,...,sysN)
[y,t] = impulse(sys)
```

Description

`impulse(sys)` plots the impulse response of an arbitrary LTI model `sys`. This model can be SISO or MIMO. The impulse response of multi-input systems is the collection of impulse responses for each input channel. The duration of simulation is determined automatically to display the transient behavior of the response.

`impulse(sys,t)` sets the simulation horizon explicitly. You can specify either a final time `t = Tfinal` (in seconds), or a vector of evenly spaced time samples of the form `t = 0:dt:Tfinal`

To plot the impulse responses of several LTI models on a single figure, use

```
impulse(sys1,sys2,...,sysN,x0)
impulse(sys1,sys2,...,sysN,x0,t)
```

When invoked with left-side arguments, the functions

```
[y,t] = impulse(sys)
[y,t,x] = impulse(sys)
y = impulse(sys,t)
```

return the output response `y`, the time vector `t` used for simulation, and the state trajectories x. No plot is drawn on the screen. For single-input systems, `y` has as many rows as time samples (`length(t)`) and as many columns as outputs. In the multi-input case, the step responses of each input channel are stacked up along the third dimension of `y`. The dimensions of `y` are then

$$(length\ of\ t) \times p \times m$$

and `y(:,:,j)` gives the response to a unit step command injected in the `j`th input channel. Similarly, the dimensions of `x` are

$$(length\ of\ t) \times n \times m$$

where n is the number of states, m the number of inputs and p the number of outputs.

3.3.3 initial()

Purpose

Plots the response of a state-space model to an initial condition.

Syntax

```
initial(sys,x0)
initial(sys,x0,t)
initial(sys1,sys2,...,sysN,x0)
[y,t,x] = initial(sys,x0)
```

Description

`initial(sys,x0)` plots the response of `sys` to an initial condition `x0` on the states. `sys` can be any state-space model (SISO or MIMO, with or without inputs). The duration of simulation is determined automatically to reflect adequately the response transients.

`initial(sys,x0,t)` explicitly sets the simulation horizon. You can specify either a final time `t = Tfinal` (in seconds) or a vector of evenly spaced time samples of the form `t = 0:dt:Tfinal`.

To plot the initial condition responses of several LTI models on a single figure, use

```
initial(sys1,sys2,...,sysN,x0)
initial(sys1,sys2,...,sysN,x0,t)
```

When invoked with left-side arguments, the functions

```
[y,t,x] = initial(sys,x0)
[y,t,x] = initial(sys,x0,t)
```

return the output response `y`, the time vector `t` used for simulation, and the state trajectories `x`. No plot is drawn on the screen. The array `y` has as many rows as time samples (`length(t)`) and as many columns as outputs. Similarly, x has `length(t)` rows and as many columns as states.

3.3.4 bode()

Purpose

Plots Bode diagram of frequency response.

Syntax

```
bode(sys)
bode(sys,w)
bode(sys1,sys2,...,sysN)
[mag,phase,w] = bode(sys)
[mag,phase] = bode(sys,w)
```

Description

`bode(sys)` plots the Bode response of an arbitrary LTI model `sys`, when invoked without left-side arguments. This model can be SISO or MIMO. In the MIMO case, `bode` produces an array of Bode plots, each plot showing the Bode response of one particular input-output channel. The frequency range is determined automatically based on the system poles and zeros.

`bode(sys,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval, set `w` to the vector of desired frequencies. Use `logspace(wmin,wmax)`[32] to generate logarithmically spaced frequency vectors. All frequencies should be specified in rad/s.

`bode(sys1,sys2,...,sysN)` plots the Bode responses of several LTI models on a single figure. All systems must have the same number of inputs and outputs. This syntax is useful to compare the Bode responses of multiple systems.

[32] see 1.3.2 logspace()

When invoked with left-side arguments, the functions

```
[mag,phase,w] = bode(sys)
[mag,phase] = bode(sys,w)
```

return the magnitude and phase (in degrees) of the frequency response at the frequencies `w` (in rad/s). The outputs mag and phase are 3-D arrays with the frequency as the last dimension.

3.3.5 margin()

Purpose

Computes gain and phase margins and associated crossover frequencies

Syntax

```
[Gm,Pm,Wg,Wp] = margin(sys)
[Gm,Pm,Wg,Wp] = margin(mag,phase,w)
margin(sys)
```

Description

`[Gm,Pm,Wc,Wp] = margin(sys)` computes the gain margin `Gm`, the phase margin `Pm`, and the corresponding crossover frequencies `Wc` and `Wp`, given the SISO open-loop model `sys`. `Wc` is the frequency where the gain is 0 dB, and `Wp` is the frequency where the phase is $-180°$.

`[Gm,Pm,Wg,Wp] = margin(mag,phase,w)` derives the gain and phase margins from the Bode frequency response data (magnitude, phase, and frequency vector). Interpolation is performed between the frequency points to estimate the margin values. This approach is generally less accurate.

When invoked without left-hand argument, `margin(sys)` plots the open-loop Bode response with the gain and phase margins marked by vertical lines. By default, gain margins are expressed in dB.

3.3.6 dcgain()

Purpose

Computes the low-frequency (DC) gain of an LTI system.

Syntax

```
k = dcgain(sys)
```

Description

`k = dcgain(sys)` computes the DC gain `k` of the LTI model `sys`.

3.3.7 nyquist()

Purpose

Produces Nyquist plot of LTI models.

Syntax

```
nyquist(sys)
nyquist(sys,w)
nyquist(sys1,sys2,...,sysN)
[re,im,w] = nyquist(sys)
[re,im] = nyquist(sys,w)
```

Description

`nyquist(sys)` plots the Nyquist response of an arbitrary LTI model `sys`. This model can be SISO or MIMO. In the MIMO case, `bode` produces an array of Nyquist plots, each plot showing the response of one particular input-output channel. The frequency range is determined automatically based on the system poles and zeros.

`nyquist(sys,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval, set `w` to the vector of desired frequencies. Use `logspace(wmin,wmax)` to generate logarithmically spaced frequency vectors. All frequencies should be specified in rad/s.

`nyquist(sys1,sys2,...,sysN)` superimposes the Nyquist plots of several LTI models on a single figure. All systems must have the same number of inputs and outputs.

When invoked with left-side arguments, the functions

```
[re,im,w] = nyquist(sys)
[re,im] = nyquist(sys,w)
```

return the real and imaginary parts of the frequency response at the frequencies `w` (in rad/sec). The return values `re` and `im` are 3-D arrays.

3.3.8 Example: Air-Dryer[33]

The air-dryer we consider here can be used to desiccate food. The air is blown through a tube and is heated at its entrance by an electric heater to a desired temperature. The heating power is proportional to the input signal $u(t)$. The heat sensor providing the output signal $y(t)$ is situated at the other end of the tube. Since the air needs a certain time until it reaches the exit, a time delay occurs in the transfer function. The nominal transfer function is given by:

$$P(s) = \frac{k_{nom}}{\tau_{nom} \cdot s + 1} e^{-\delta_{nom} \cdot s}$$

The weighting function $W1(s)$ and the uncertainty bound $W2(s)$ are also given:

$$W1(s) = 100 \frac{(1.6 \cdot s + 1)^2}{(15 \cdot 1.6 \cdot s + 1)^2}, W2(s) = 0.25 \frac{0.6 \cdot s + 1}{0.18 \cdot 0.6 \cdot s + 1} \cdot (0.06 \cdot s + 1)^2$$

In this example we will try to derive a controller with the Aström-Hägglund approach. After some loop-shaping, we will check if the controlled system satisfies the robust performance condition. To conclude, the step response of the closed-loop system will be displayed.

M-File

```
clear all, clc, close all

% Parameters:
k_nom=2.99;
delta_nom=0.27;
tau_nom=0.63;

% Transfer Function:
s=tf('s');
P=k_nom/(tau_nom*s+1)*exp(-delta_nom*s);

% Astroem-Haegglund:
P0 = dcgain(P)                  % DC gain
[Gm,Pm,Wg,Wp] = margin(P)
kp_krit = Gm;                   % Critical gain
T_krit = 2*pi/Wg;               % Critical time constant
kappa = inv(P0*kp_krit);

% Astroem-Haegglund Parameters (mu=0.50):
alpha0_kp = 0.13; alpha1_kp = 1.9; alpha2_kp = -1.3;
alpha0_Ti = 0.9; alpha1_Ti = -4.4; alpha2_Ti = 2.7;

% Astroem-Haegglund Controller:
kp = kp_krit*(alpha0_kp*exp(alpha1_kp*kappa+alpha2_kp*kappa^2));
Ti = T_krit*(alpha0_Ti*exp(alpha1_Ti*kappa+alpha2_Ti*kappa^2));

% Controller:
Ta=2; aa=6.2;                   % Lag element parameters
Tb=0.01;                        % Low pass parameter
```

[33] see chapter 15, Analysis and Synthesis of SISO Control Systems

```
Tc=1.8; ac=0.3;                % Lead element parameters
k=1.5;                         % Additional gain
C=kp*(1+1/(Ti*s))*(Ta*s+1)/(aa*Ta*s+1)*1/(Tb*s+1)*k*(Tc*s+1)/(ac*Tc*s+1);

% Discrete loop gain
L=C*P;
figure(1)
margin(L)                      % Plot the Bode plot and check phase margin
w = logspace(-3,3,1e4);        % Frequency vector [rad/s]
[re,im]=nyquist(L,w);

% Robust Performance Condition:
S=1./(1+re+i*im);                                  % Discrete sensitivity
T=(re+i*im)./(1+re+i*im);                          % Discrete complementary sensitivity
s=tf('s');
W1=100*(1.6*s+1)^2/(15*1.6*s+1)^2;                 % Weighting function
W2=0.26*(0.65*s+1)/(0.24*0.65*s+1)*(0.06*s+1)^2;   % Uncertainty bound

magS=abs(S);
magT=abs(T);
[magW1,phW1]=bode(W1,w);
[magW2,phW2]=bode(W2,w);
Wsmag = magW1.*magS+magW2.*magT;                   % Robust performance value
figure(2)
semilogx(w,squeeze(Wsmag),'r',w,ones(size(w)),':k') % Robust performance condition
xlabel('Frequency [rad/s]')
ylabel('Magnitude [dB]')
axis([10^-3,10^3,0,1.5])
```

Output:

```
P0 =

2.9900

Gm =

1.4470

Pm =

40.3465

Wg =

6.6816

Wp =

4.4727
```

Explanation

We begin by defining the system parameters and the transfer function. In order to apply the Aström-Hägglund control design, some parameters of the system plant have to be computed. The DC gain `P0` can be obtained using the function `dcgain()`[34]. The function `margin()`[35] returns the minimum gain margin `Gm`, the phase margin `Pm` and the associated crossover frequencies `Wg` and `Wp` of the

[34] see 3.3.6 dcgain()
[35] see 3.3.5 margin()

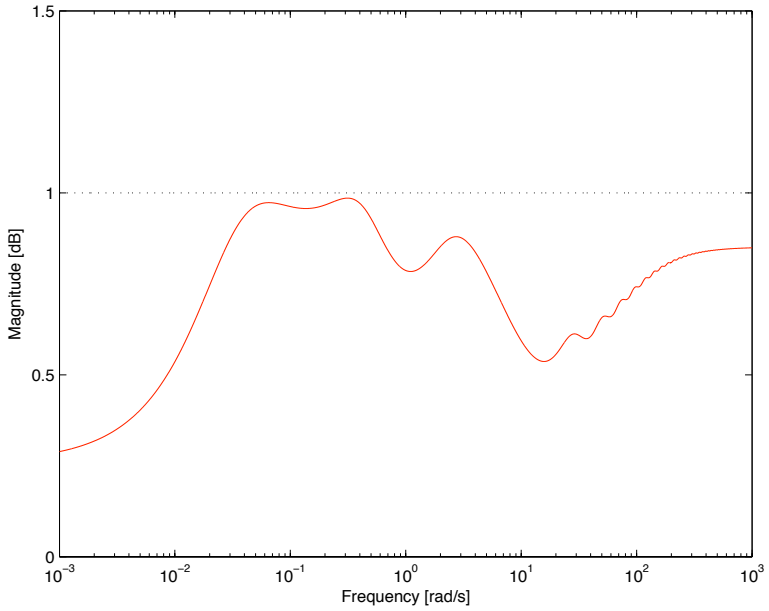


Fig. 3.3. Robust performance condition for the air-dryer with controller

open-loop system. Subsequently, the system values `kp_krit`,`T_krit` and `kappa` can be computed. After defining the Aström-Hägglund parameters ($\mu = 0.50$) we evaluate `kp` and `Ti`. Additionally to the Aström-Hägglund PI-controller we add a lag, a lead and a low-pass element to improve the system properties.

First we check whether the Bode plot suits our conceptions. Therefore, we use the function `margin()`, which additionally computes the phase margin and displays it in the Bode plot. The plot is depicted in Fig. 3.4

With the command `w=logspace(-3,3)`[36], we generate a logarithmically spaced vector `w` from 10^{-3} to 10^{3}. Because Matlab can't directly operate with systems with time delay, we discretise the loop gain transfer function with the command `[re,im]=nyquist(L,w)` using the frequency vector `w`. With the real part vector `re` and the imaginary part vector `im`, we can compute the discrete sensitivity `S` and the discrete complementary sensitivity `T`.

To check if the robust performance conditions are satisfied, we also need the weighting function `W1` and the uncertainty bound `W2`. The following `abs()` and `bode()`[37] functions compute the magnitude and the phase of `S`,`L`,`W1` and `W2` at the frequencies stored in the vector `w`. The robust performance condition is met if the equation $\left\| |W_1(s) \cdot S(s)| + |W_2(s) \cdot T(s)| \right\|_\infty < 1$ is satisfied. The

[36] see 1.3.2 logspace()
[37] see 3.3.4 bode()

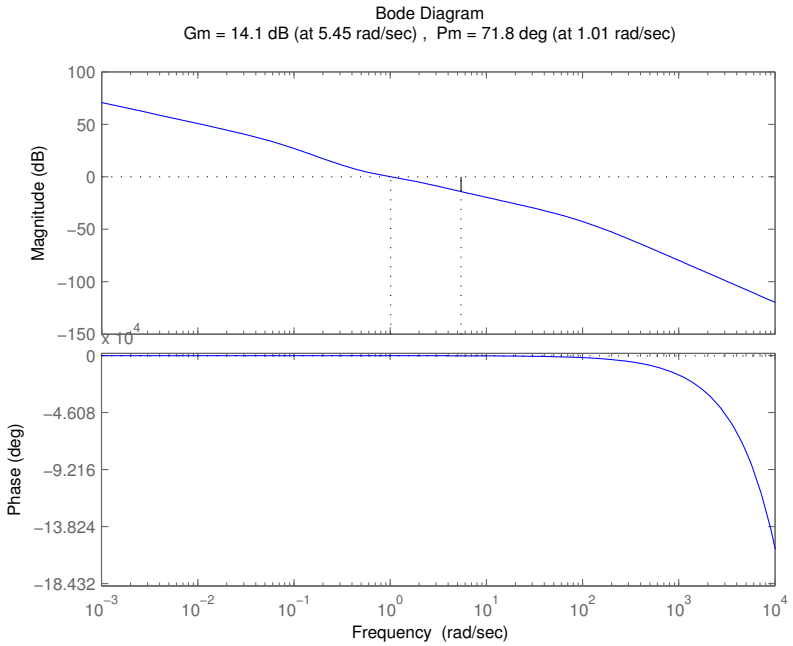


Fig. 3.4. Bode plot of the controlled air-dryer with phase margin

evaluation of the left-hand side of the equation at the frequencies `w` is saved in the vector `Wsmag`. The command `semilogx` plots `Wsmag` versus `w` in red and a horizontal line at $y = 1$ in black, dotted style. Note that we use the function `squeeze` to extract the data from the three-dimensional array `Wsmag` to a vector. The instructions `xlabel` and `ylabel` label the x and the y axis, while `axis` sets the frame border. Fig. 3.3 shows that robust performance conditions are met.

In the three last command lines we plot the step and the impulse responses in the same figure. For both functions `step()`[38] and `impulse()`[39], we add the time-delay to the closed-loop transfer function. With the second argument we define the time span and the sample time.

3.3.9 Example: Geostationary Satellite [40]

Due to the fact that the satellite is unstable, we have to design a controller. During the design process we have to check how the controlled satellite reacts to various external influences such as a hit by a small meteorite.

[38] see 3.3.1 step()
[39] see 3.3.2 impulse()
[40] see chapter 3.11, Analysis and Synthesis of MIMO Control Systems

M-File

```
clear all, clc,close all

% Parameters:
r0=4.22*10^7;                   % Set value for the orbit radius [m]
w0=7.29*10^-5;                  % Set value for the angular velocity radius [rad/s]

% System Matrices
A=[0 1 0 0;3*w0^2 0 0 2*r0*w0;0 0 0 1;0 -2*w0/r0 0 0];
B=[0 0;1 0;0 0;0 1/r0];
C=[1/r0 0 0 0;0 0 1 0];
D=zeros(2);

% LQR:
Q = C'*C;
R = 4*10^-5*eye(2);
K = lqr(A,B,Q,R);               % Optimal gain matrix

% Open-Loop transfer Function:
sys_ol=ss(A,B,K,D);
figure(1)
sigma(sys_ol)                                               % Plot the singular values
line([10^-5,10^-1],[0,0],'LineStyle',':','Color','black')  % Draw a dotted line at 0 dB
axis([10^-5,10^-1,-40,60])                                  % Set frame

% Closed Loop:
sys_cl=ss(A-B*K,B,C,D);
figure(2)
initial(sys_cl,[0 -r0 0 0]')    % Plot the initial condition response
```

Output:

Explanation

After the definition of the system matrices, we design a controller using `lqr()`[41]. With the new controller, we derive the open-loop transfer function `sys_ol` using `ss(A,B,K,D)`[42]. Please note that we use `K` instead of `C` to include the new LQR controller in our loop.

The singular value plot of the open-loop transfer function seen in Fig. 3.5 was generated using the function `sigma(sys_ol)`[43]. The plot shows us that the maximal and minimal singular values are very close to each other at the crossover frequency. That tells us that we chose good parameters for our LQR controller because the controlled system shows an 'almost SISO' behaviour at the crossover frequency. We use the command `figure()` to tell MATLAB in which window the plot should be drawn. The parameter `line([10^-5,10^-1],[0,0],'LineStyle',':','Color','black')` draws a horizontal line in the plot. `[10^-5,10^-1]` defines the starting and ending x-coordinate, `[0,0]` the corresponding y-coordinates. The additional parameters define the dotted line style and the black colour. Please note that for additional drawings, we use the values as they are on the y-axis (in dB). The command `axis([10^-5,10^-1,-40,60])` sets the border of the plot.

[41] see 3.4.3 lqr()
[42] see 2.1.1 ss()
[43] see 3.2.3 sigma()

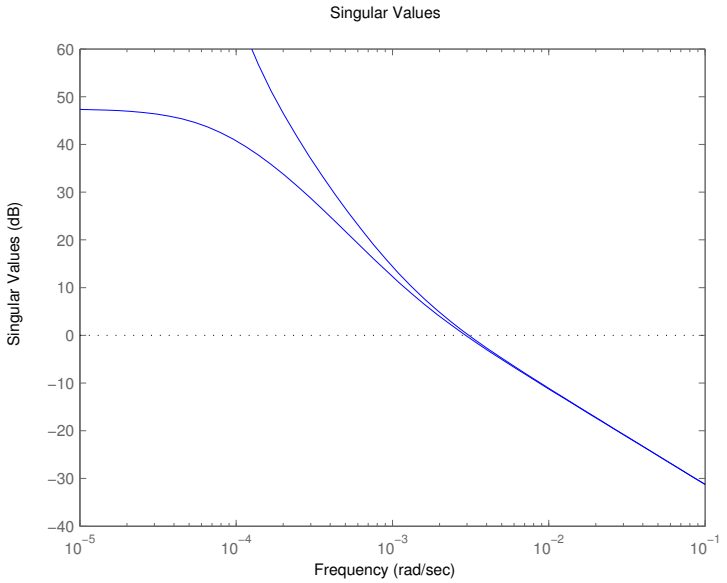


Fig. 3.5. Singular values of the satellite with LQR controller

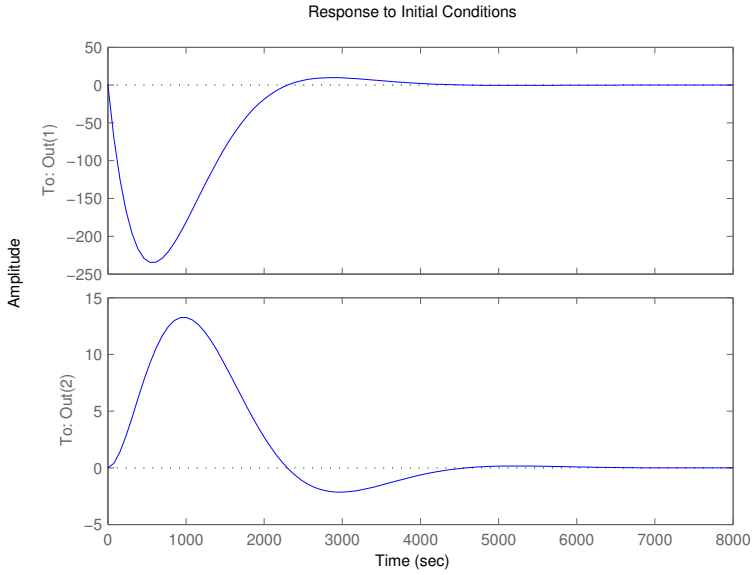


Fig. 3.6. Response to – 1 m/s initial radial velocity

To investigate the closed loop system behaviour, we build the transfer function `sys_cl` using `ss(A-B*K,B,C,D)`[44] . We now simulate a hit from a small meteorite using `initial(sys_cl,[0 -r0 0 0]')`[45]. Therefore, we set the initial radial velocity to `-r0`, which corresponds to $-1m/s$ (due to normalization). As one can derive from Fig. 3.6, the time required until the initial deviation is corrected is about 3000 seconds. The benefits, on the other hand, are the small fuel consumption of the thrusters, which leads to a longer lifespan of the satellite. Therefore, the controller design is acceptable.

3.4 Control Design

With the exponential growth of computing power, several new tools for controller design have been developed. Suddenly, large systems of equations can be solved in a snatch. Iterative optimization methods, such as `fminsearch()`[46] have become popular. The command `lqr()`[47] solves a linear system of equation to derive the optimal gain matrix.

When the root-locus method was first introduced, there was not enough computing power to get the exact path of the poles and zeros. Therefore, a large set of rules were derived to draw approximate root-locus diagrams. Nowadays, most of these rules are obsolete and have been replaced by numerical algorithms such as `rlocus()`[48].

3.4.1 rlocus()

Purpose

Plots the root locus of LTI models

Syntax

```
rlocus(sys)
rlocus(sys1,sys2,...)
[r,k] = rlocus(sys)
r = rlocus(sys,k)
```

[44] see 2.1.1 ss()
[45] see 3.3.3 initial()
[46] see 3.4.2 fminsearch()
[47] see 3.4.3 lqr()
[48] see 3.4.1 rlocus()

Description

`rlocus(sys)` computes the root locus of the SISO model `sys`. The root locus gives the closed-loop pole trajectories as a function of the feedback gain k (assuming negative feedback). Root loci are used to study the effects of varying feedback gains on closed-loop pole locations. In turn, these locations provide indirect information on the time and frequency responses.

When invoked with output arguments,

```
[r,k] = rlocus(sys)
r = rlocus(sys,k)
```

return the vector `k` of selected gains and the complex root locations `r` for these gains. The matrix `r` has `length(k)` columns, and its `j`th column lists the closed-loop roots for the gain `k(j)`.

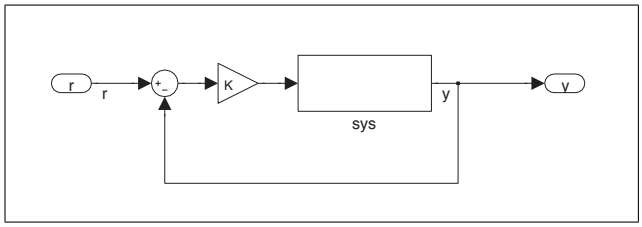


Fig. 3.7. Root Locus System Setup

3.4.2 fminsearch()

Purpose

Find minimum of unconstrained multivariable function

Syntax

```
x = fminsearch(@fun,x0)
```

Description

`x = fminsearch(@fun,x0)` starts at the point `x0` and finds a local minimum `x` of the function described in `@fun`. The input value `x0` can be a scalar, vector, or matrix whereas `@fun` is a function handle.

3.4.3 lqr()

Purpose

Calculates a linear-quadratic (LQ) state-feedback regulator for state-space system

Syntax

```
K=lqr(A,B,Q,R)
K=lqr(sys,Q,R)
L=(lqr(A',C',B*B',q))'
```

Description

The function `K=lqr(A,B,Q,R)` calculates the optimal gain matrix `K` such that, for a continuous time system, the state-feedback law $u = -Kx$ minimizes the quadratic cost function

$$J(u) = \int_0^\infty [x^T(u(t)) \cdot Q \cdot x(u(t)) + u^T(t) \cdot R \cdot u(t)]dt$$

subject to the system dynamics $\dot{x} = Ax + Bu$. The resulting block diagram can be found in Fig. 3.8.

The variables Q and R can be seen as "tuning knobs". Setting $R = r \cdot I_{m \times m}, r > 0$ often yields good results.

The matrices must satisfy the conditions:

$$Q = Q^T \in \mathbb{R}^{n \times n}, Q \geq 0,$$

and

$$R = R^T \in \mathbb{R}^{m \times m}, Q > 0,$$

where n is the dimension of A and m the number of columns of B (obviously for SISO systems, $m = 1$)

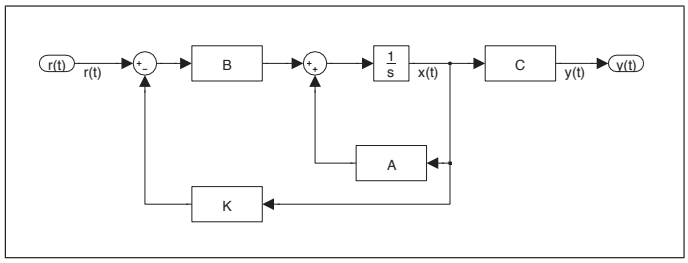


Fig. 3.8. LQR Block Diagram

The command `L=(lqr(A',C',B*B',q))'` computes the observer gain `L` for an LQG controller. Varying `q` changes the behaviour of the LQG controller.

3.4.4 Example: Air-Dryer [49]

To show how to use `fminsearch()`[50], we will consider the air-dryer[51] again. We will try to optimize the previously derived controller. Therefore we introduce the objective function `J` which includes various parameters describing the performance of our controlled system. It incorporates the square of the error signal, the maximum overshoot and the robust performance condition.

Two M-Files are built. The 'master' M-File `AHmaster.m` defines the problem and the initial parameters, calls the optimization function `fminsearch()` and displays the result. The command `fminsearch()` repeatedly calls the function
`AHSys_fmin(par)` defined in the second M-file, which basically returns the value of the objective function `J`. To compute the performance parameters, we use a Simulink model of the system `AHSys.mdl`. The system setup is depicted in Fig. 3.9.

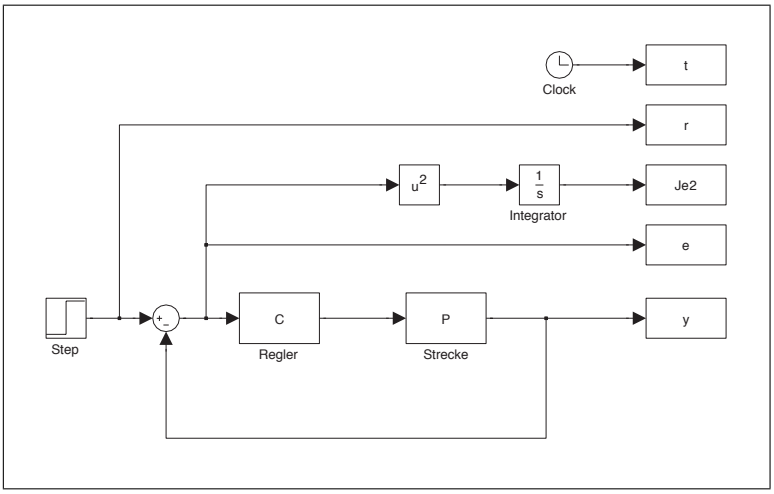


Fig. 3.9. Block Diagram of AHSys

M-File (AHmaster.m)

```
clc, clear all, close all

% Define global variables:
global C s tsim mu1 mu2 mu3 P w kp Ti magW1 magW2
```

[49] see chapter 15, Analysis and Synthesis of SISO Control Systems
[50] see 3.4.2 fminsearch()
[51] see 3.3.8 Example: Air-Dryer

```
% Parameters:
k_nom=2.99;
delta_nom=0.27;
tau_nom=0.63;

% Transfer functions:
s=tf('s');
P=k_nom/(tau_nom*s+1)*exp(-delta_nom*s);          % Nominal plant transfer dunction
W1=100*(1.6*s+1)^2/(15*1.6*s+1)^2;                % Weighting function
W2=0.26*(0.65*s+1)/(0.24*0.65*s+1)*(0.06*s+1)^2; % Uncertainty bound
w = logspace(-3,3,1e3);                           % Frequency vector [rad/s]
[magW1,phW1]=bode(W1,w);
[magW2,phW2]=bode(W2,w);

% Astroem-Haegglund PI-parameter:
kp = 0.2723;
Ti = 0.3536;

% Initial parameters for function to be optimized:
mu1 = 1;                                          % Weight of the error signal
mu2 = 1;                                          % Weight of the maximum overshoot
mu3 = 100;                                        % Weight for the robust performance condition
var10 =0.01;                                      % Low pass parameter
var20= 2; var30= 6.2;                             % Lag element parameters
var40= 1.8; var50= 0.3;                           % Lead element parameters
k0=kp*1.5;                                        % Gain
tsim = 20;                                        % Simulation time [s]

% Optimization:
par0=[var10;var20;var30;var40;var50;k0];
par_opt=fminsearch(@AHSys_fmin,par0);             % Optimized parameters

% Get optimal parameters:
var1=par_opt(1)
var2=par_opt(2)
var3=par_opt(3)
var4=par_opt(4)
var5=par_opt(5)
k=par_opt(6)

% Simulation with optimized controller:
C_opt = k*(1+1/(Ti*s))*(1/(var1*s+1))*(var2*s+1)/(var2*var3*s+1)*(var4*s+1)/(var4*var5*s+1);
C=C_opt;
sim('AHSys',tsim)
y_opt = y;

% Simulation with Astroem-Haegglund controller:
C_AH = k0*(1+1/(Ti*s))*(1/(var10*s+1))*(var20*s+1)/(var20*var30*s+1)*(var40*s+1)/(var40*var50*s+1);
C=C_AH;
sim('AHSys',tsim)
y_AH = y;

% Plot step responses:
figure
plot(t,y_opt,'k',t,y_AH,'--k',t,r,'--r')
xlabel('Time [s]')
ylabel('r,y')
legend('Optimal','Astroem-Haegglund','Location','SouthEast')

% Plot Nyquist diagram:
figure
nyquist(P*C_opt,'k',P*C_AH,'--k')
axis([-1,1,-2,2])
axis equal
legend('Optimal','Astroem/Haegglund','Location','East')

% Check robustness of optimized controller:
```

```
L_opt=C_opt*P;                                  % Loop gain with optimized controller
[re,im]=nyquist(L_opt,w);
S=1./(1+re+i*im);                               % Discrete sensitivity with optimized controller
T=(re+i*im)./(1+re+i*im);                       % Discrete complementary sensitivity with optimized controller
magS=abs(S);
magT=abs(T);
Wsmag = magW1.*magS+magW2.*magT;                % Robust performance value with optimized controller
figure
semilogx(w,squeeze(Wsmag(:,:,:)),'k'),hold on

% Check robustness of AH controller:
L_AH=C_AH*P;                                    % Loop gain with AH controller
[re,im]=nyquist(L_AH,w);
S=1./(1+re+i*im);                               % Discrete sensitivity with AH controller
T=(re+i*im)./(1+re+i*im);                       % Discrete complementary sensitivity with AH controller
magS=abs(S);
magT=abs(T);
Wsmag = magW1.*magS+magW2.*magT;                % Robust performance value with AH controller
semilogx(w,squeeze(Wsmag(:,:,:)),'--k',w,ones(size(w)),'--r')
xlabel('Frequency [rad/s]')
ylabel('Magnitude')
axis([10^-3,10^3,0,1.5])
legend('Optimal','Astroem/Haegglund','Location','NorthEast')
```

M-File (AHSys_fmin.m)

```
function J = AHSys_fmin(par)

%Global variables:
global C s tsim mu1 mu2 mu3 P w Ti magW1 magW2

%Parameters:
var1 = par(1);                                  % Low pass parameter
var2 = par(2); var3 = par(3);                   % Lag element parameters
var4 = par(4); var5 = par(5);                   % Lead element parameters
k = par(6);                                     % Gain

%Simulation:
C = k*(1+1/(Ti*s))*(1/(var1*s+1))*(var2*s+1)/(var2*var3*s+1)*(var4*s+1)/(var4*var5*s+1);
sim('AHSys',tsim)                               % Simulation
L = C*P;                                        % Loop gain
[re,im]=nyquist(L,w);
S=1./(1+re+i*im);                               % Discrete sensitivity
T=(re+i*im)./(1+re+i*im);                       % Discrete complementary sensitivity
magS=abs(S);
magT=abs(T);
Wsmag = magW1.*magS+magW2.*magT;                % Robust performance value

%Objective function:
J = mu1*Je2(end)+mu2*(max(y)-1)+mu3*(max(Wsmag)>0.99);
```

Output:

```
kp = 0.2692

Ti = 0.3640

var1 = 0.0996

var2 = 100.4017

var3 = 0.7615
```

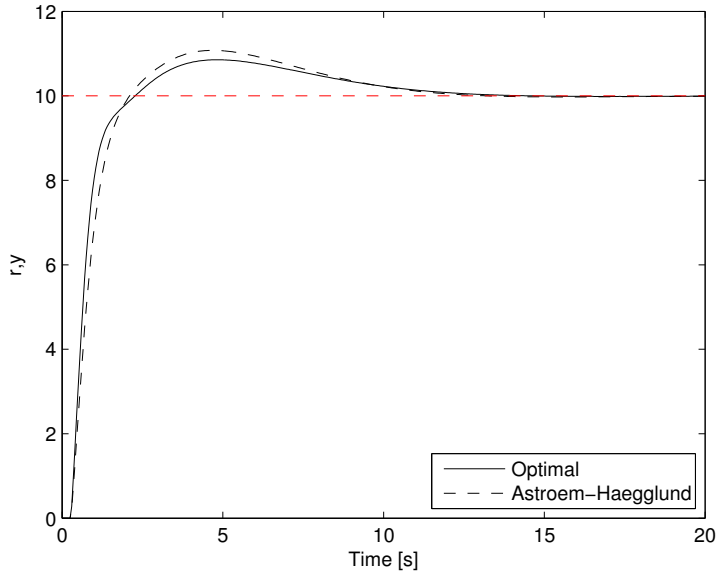


Fig. 3.10. Step response of the closed loop systems

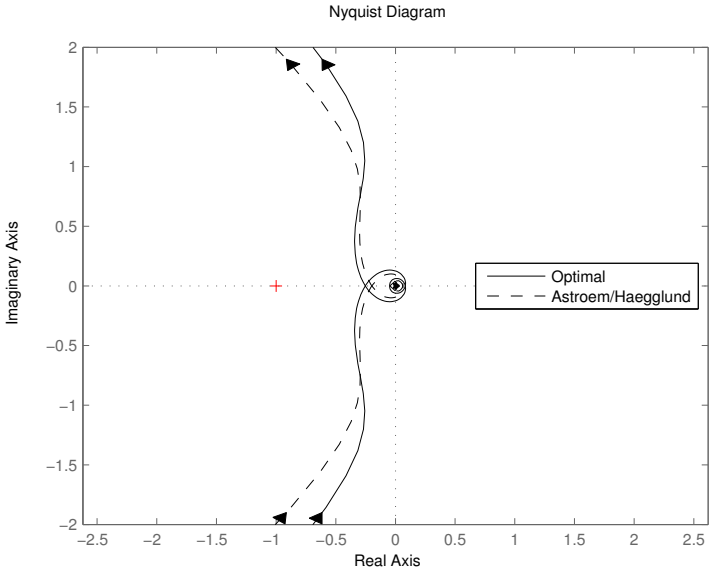


Fig. 3.11. Nyquist diagram of the open-loop systems

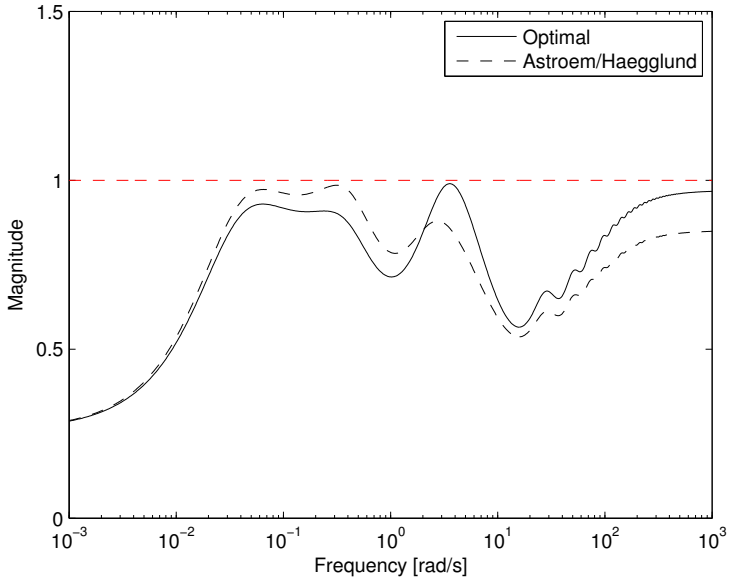


Fig. 3.12. Robust performance condition for both systems

Explanation

In the first M-File we begin by defining the variables `C`, `s`, `tsim`, `mu1`, `mu2`, `mu3`, `P`, `w`, `Ti`, `magW1` and `magW2` as global variables, so we can access their values from everywhere in the Matlab environment. We also determine the system parameters and the transfer functions of `P`, `W1` and `W2`. With the function `bode()`[52] we compute the magnitudes `magW1` and `magW2`. As initial guess (`var10`,`var20`,`var30`,`var40`,`var50`,`k0`) for `fminsearch()`[53], we use the controller which was designed in *3.3.8 Example: Air-Dryer*. The variables `mu1`, `mu2` and `mu3` describe the weights of the square of the error signal, the maximum overshoot and the robust performance condition, respectively. The command `tsim` is the time span of the simulation. Having set all these parameters, we can call the function `fminsearch()`. The optimisation takes a while; it determines iteratively the local minimum of the objective function `J`.

With the returned optimized values we build the controller `C_opt`. The variable `C_AH` represents the controller we found in *3.3.8 Example: Air-Dryer*. Next, the step responses of both controlled system are simulated with the model `AHSys`. The vectors `y_opt` and `y_AH` contain the returned values. These are plotted with the command `plot(t,y_opt,'k',t,y_AH,':k',t,r,'--r')`.

[52] see 3.3.4 bode()
[53] see 3.4.2 fminsearch()

Also the Nyquist diagrams are displayed, using `nyquist()`[54]. The plot and the Nyuist diagram are depicted in Fig. 3.10 and Fig. 3.11, respectively.

Just as in *3.3.8 Example: Air-Dryer*, the robust performance condition is checked with both the controllers `C_opt` and `C_AH`. Fig. 3.12 shows that it is still satisfied.

The first command line of the second M-File defines `AHSys_fmin()` as a new function, where `par` is the only argument of the function and `J` is the returned value. After defining the global variables, we can build the controller `C` using the 6 values in the array `par`. The command `sim('AHSys',tsim)`[55] simulates the controlled system with the model `AHSys`. Because Matlab can't properly manipulate systems with time delay, we have to discretize the open-loop transfer function `L`. Therefore we first use the `nyquist()`[56] function to get the real part vector `re` and the imaginary part vector `im`. The discrete sensitivity `S` and the discrete complementary sensitivity `T` can then be computed. Mathematically, this can be formulated as

$$S(j\omega) = \frac{1}{1 + L(j\omega)} = \frac{1}{1 + re^{j\varphi}}, r = |L(j\omega)|, \varphi = \angle L(j\omega)$$

$$T(j\omega) = \frac{L(j\omega)}{1 + L(j\omega)} = \frac{re^{j\varphi}}{1 + re^{j\varphi}}, r = |L(j\omega)|, \varphi = \angle L(j\omega)$$

The output value `magS` is the magnitude of the sensitivity `S` and `magT` is the magnitude of the complementary sensitivity `T`.

Finally, the objective function `J` can be computed by multiplying each weighting factor with its performance parameter. The square of the error signal is evaluated during the simulation of the model `AHSys` and stored in the variable `Je2`. The maximum overshoot is the maximal value of the step response `y` minus `1`. The robust performance condition can be obtained using the different magnitude vectors. The command `max(Wsmag)>0.99` returns the value `1` if `Wsmag` is larger than `0.99`. Using a large weighting `mu3`, we can assure that the condition is met. In the case where the condition is not satisfied (`Wsmag` is larger than `0.99`) the objective function increases significantly.

The command `fminsearch()`[57] computes the value of `J` several times in an attempt to find a local minimum.

[54] see 3.3.7 nyquist()
[55] see 4.4 sim()
[56] see 3.3.7 nyquist()
[57] see 3.4.2 fminsearch()

3.4.5 Example: Levitating Sphere [58]

In order to demonstrate the functionality of the function `lqr()`[59], we will pick up *3.2.8 Example: Levitating Sphere*. An LQG controller is to be designed.

M-File

```
clear all, clc, close all

%State-space matrices:
A=[0 1 0;700 0 700;0 0 -0.2];
B=[0 0 1]';
C=[1 0 0];
D=0;

%Save as a state-space object:
sys_ss = ss(A,B,C,D);

%LQR:
Q = C'*C;
R = 1;                                          % Tuning for LQR
K = lqr(A,B,Q,R);                               % Optimal gain matrix

%Open-Loop LQR transfer Function:
sys_LQR=ss(A,B,K,D);

%Open-Loop LQG transfer Function using LTR for q=10^-7,10^-10,10^-13:
for i=7:3:13
    q=10^(-i);                                  % Tuning for LQG
    L=lqr(A',C',B*B',q)';                       % Optimal observer matrix
    sys_LQG=ss(A-B*K-L*C,-L,-K,D)*ss(A,B,C,D); % Open-Loop state-space model
    axis([-2,0.2,-1.2,1.2])
    axis equal;
    nyquist(sys_LQG),hold on                    % Nyquist plot of the 3 LQG systems
end
nyquist(sys_LQR,'--'),hold on                   % Nyquist plot of the LQR system
legend('LQG q=1E-7','LQG q=1E-10','LQG q=1E-13','LQR','location','SouthEast')
```

[58] see chapter 3.10, Analysis and Synthesis of MIMO Control Systems
[59] see 3.4.3 lqr()

Output:

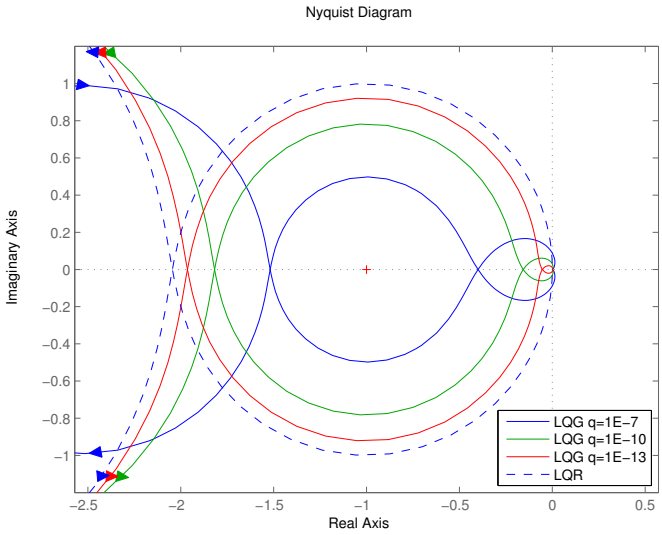


Fig. 3.13. Nyquist plots of LQR and LQG

Explanation

The first few command lines define the parameters and the system. The variable `Q` can be defined as `C'*C`, and for `R` we choose `1`. The optimal gain matrix `K` is the return value of `lqr(A,B,Q,R)`[60]. Using `ss()`[61] we obtain the corresponding open-loop LTI model. In a for-loop, we calculate the observer gain matrix `L` for three different values of the tuning parameter q $(10^{-7}, 10^{-10}, 10^{-13})$. In each loop, we compute `L`, build the open-loop model and display the Nyquist diagram. In conclusion we display the Nyquist plots of the LQR model in Fig. 3.13 . We can observe that the LQG Nyquist plots approach the LQR Nyquist plot for decreasing values of q.

[60] see 3.4.3 lqr()
[61] see 2.1.1 ss()

4

Simulink

4.1 Introduction

Simulink is a MATLAB toolbox for modeling, simulating, and analysing dynamic systems. It supports linear and nonlinear systems, modeled in continuous time and sampled time.

Simulink enables you to pose a question about a system, model it, and see what happens. With Simulink, you can easily build models from scratch.

For modeling, Simulink provides a graphical user interface (GUI) for building models as block diagrams, using click-and-drag mouse operations. Simulink includes a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors.

After you define a model, you can simulate it, using a choice of mathematical integration methods, either from the Simulink menus or by entering commands in the Command Window. Using scopes and other display blocks, you can see the simulation results while the simulation is running. The simulation results can be put in the MATLAB workspace for post-processing and visualization.

4.2 Working with Simulink

4.2.1 Defining a New Model

To open a new model in MATLAB, click on the pulldown menu File → New → Model. A new window opens. You should now save the empty model in the same directory as the M-File. That simplifies manipulating the model from the M-File. MATLAB sets all settings for the simulation to a reasonable value; we can leave them as they are. One exception must be made: We have to set the 'Automatic solver parameter selection' from *warning* to *none*. You find this option in the model window in the pulldown menu Simulation → Configurations Parameters → Diagnostics. Without this setting, a warning will be displayed every time you simulate the model.

4.2.2 Adding and Connecting Blocks

We can now drag new models into the blank space. The models are stored in the *Library Explorer* which can be found in the pulldown menu View → Library Browser[1]. We chose a *Sine Wave*[2] input from the *Library Browser* submenu *Sources* and drag it to the empty workspace. As a next step, we drag an *Integrator*[3] from the submenu *Continuous* to the workspace. The last element we add is a gain element from *Math Operations*. By double-clicking it, we open a window where we can define its gain. We set it to `mult`. Later on, when running the simulation, the model will look up the variable `mult` in the corresponding M-file.

To get the simulated data back into the MATLAB workspace we add two *To Workspace*[4] blocks from the submenu *Sinks*. By double-clicking it, we define the variable name of the output. Additionally, we change the *Save format* to *Array* for easier manipulation of the output variables in the MATLAB workspace after simulation.

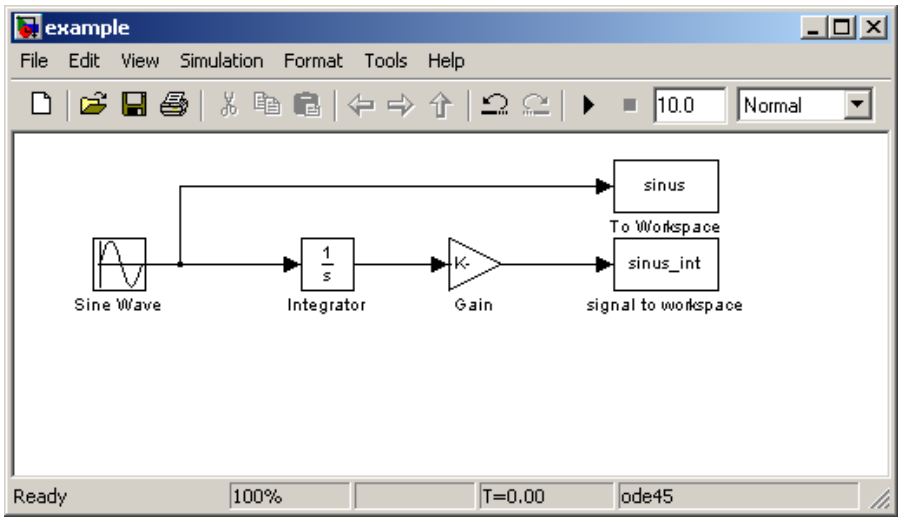


Fig. 4.1. Block diagram in Simulink window

Now we connect the blocks in the order as shown in Fig. 4.1. You can either drag an arrow from the source block to the target block or click on

[1] The Library looks significantly different on Macs, the functional range however stays the same
[2] see 4.5.3 Sine Wave
[3] see 4.6.2 Integrator
[4] see 4.11.3 To Workspace

the source block and then ctrl-click on the target block. To get a branching, ctrl-click the arrow at the desired point.

4.2.3 Input-Output

To get the simulation running, we write an M-file which will be saved in the same directory as the MDL-file.

M-file

```
clear all,clc, close all
% Set gain factor
mult=0.5;
% Set simulation timespan
tsim=10;
% Simulating example.mdl
sim('example',tsim);
plot(tout,sinus,tout,sinus_int,'--');
xlabel('Time [s]');
ylabel('Magnitude');
legend('sinus','sinus_int','Location','SouthEast')
```

Output:

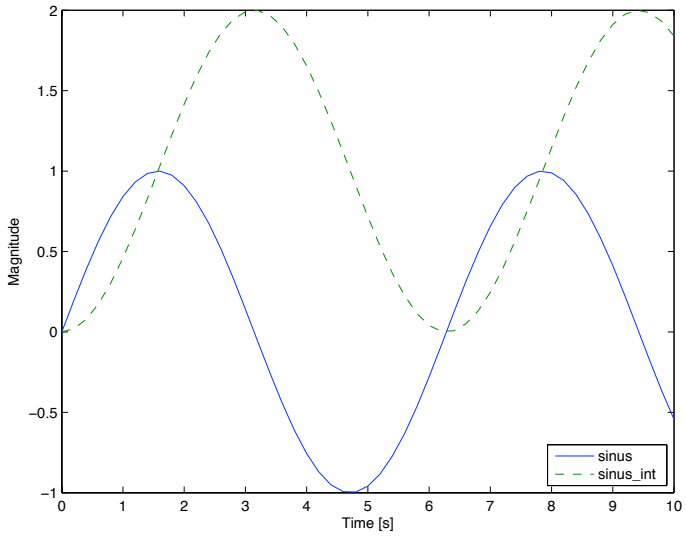


Fig. 4.2. Simulation Output

Explanation

After the setting of the the gain factor `mult` and the time span `tsim`, we run the simulation of `example.mdl` using `sim('example',tsim)`[5]. The vector `tout` contains all time steps for which the simulation is being evaluated. This vector is returned by default after every simulation. Its name can be changed in the submenu Simulation → Configuration Parameters → Data Import/Export.

The returned vectors `sinus` and `sinus_int` contain the simulated data. More specifically; `sinus_int(i)` holds the value of the signal at the corresponding time stored in `tout(i)`.

The plot of `sinus` and `sinus_int` versus `tout` is depicted in Fig. 4.2

4.3 Tips and Tricks

- You can rotate blocks by pressing ctrl-R.
- Connecting blocks can be facilitated by clicking on the source block and then ctrl-clicking the target block.
- You can get a branch by ctrl-clicking the arrow.
- Encapsulate big systems into smaller subsystems: The controller and the plant belong in a *Subsystem*[6] block for the sake of abstraction.
- Always make sure that you chose the right properties using the *Gain*[7] block. The output differs significantly depending on *element-wise* or *matrix* multiplication!
- Every arrow must end in a block. Unnecessary outputs must be connected to the *Terminator*[8] block.
- Don't forget to change the *Save Format* of a *To Workspace*[9] block to *Array*.
- The initial value of an internal state can be set in the *Integrator*[10] for non-linear models and in the *State-Space*[11] box for linearized systems.
- By double-clicking the signal arrow, you can define its name.
- If the length of the output vectors differs from `tout`, uncheck the box *Limit data points to last* found in Simulation → Configurations Parameters → Data Import/Export.

[5] see 4.4 sim()
[6] see 4.9.3 Subsystem
[7] see 4.8.1 Gain
[8] see 4.11.2 Terminator
[9] see 4.11.3 To Workspace
[10] see 4.6.2 Integrator
[11] see 4.6.3 State-Space

4.4 sim()

Purpose

Simulates a dynamic system.

Syntax

```
sim(model,timespan)
```

Description

The `sim()` command executes a Simulink model, using all Configuration Parameters dialog box settings, including the options specified in the Data Import/Export pane.

In the Configuration Parameters dialog box, there are various settings concerning the values of a model's parameters, such as solver type and simulation start or stop time. In the Data Import/Export pane you can specify which data is imported and exported to the MATLAB workspace.

4.5 Sources

4.5.1 Clock

Description

The Clock block outputs the current simulation time at each simulation step. This block is useful for other blocks that need the simulation time.

4.5.2 Constant

Description

The Constant block generates a constant scalar, vector or matrix output.

Parameters

- *Constant value*: Specifies the ouput of the block. You can enter any MATLAB expression in this field.

4.5.3 Sine Wave

Description

The Sine Wave block provides a sinusoid. The output is determined by:

$$y = Amplitude \cdot sin(frequency \cdot time + phase) + bias$$

Parameters

- *Amplitude*: The amplitude of the signal. The default is `1`.
- *Bias*: Constant value added to the sine to produce the output of this block.
- *Frequency*: The frequency, in radians/second. The default is `1` rad/s.
- *Phase*: The phase shift, in radians. The default is `0` radians.

4.5.4 Step

Description

The Step block provides a step between two definable levels at a specified time.

Parameters

- *Step time*: The time, in seconds, when the output jumps from the *Initial value* parameter to the *Final value* parameter. The default is `1` second.
- *Initial value*: The block output until the simulation time reaches the *Step time* parameter. The default is `0`.
- *Final value*: The block output when the simulation time reaches and exceeds the *Step time* parameter. The default is `1`.

4.6 Continuous

4.6.1 Derivative

Description

The Derivative block approximates the derivative of its input. The initial output for the block is zero.

4.6.2 Integrator

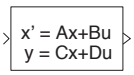
Description

The Integrator block outputs the integral of its input. The initial output of the block can be defined as a parameter on the block dialog box or can be imported them from an external signal. In the second case an additional input port appears under the block input.

Parameters

- *Initial condition source*: If set to `internal`, gets the states' initial conditions from the *Initial condition* parameter. If set to `external`, it gets them from an external block.
- *Initial condition*: The states' initial conditions.

4.6.3 State-Space



Description

The State-Space block implements a system whose behaviour is defined by:

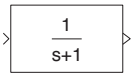
$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

where x is the state vector, u is the input vector, and y is the output vector.

Parameters

- *A,B,C,D*: The matrix coefficients.
- *Initial condition*: The initial state vector.

4.6.4 Transfer Function



Description

The Transfer Function block models a linear system by a transfer function of the Laplace-domain variable s. This block assumes that the transfer function has the following form:

$$h(s) = \frac{y(s)}{u(s)} = \frac{b_m s^m + b_{m-1} s^{m-1} + \ldots + b_2 s^2 + b_1 s + b_0}{a_n s^n + a_{n-1} s^{n-1} + \ldots + a_2 s^2 + a_1 s + b_0}$$

where u is the input vector and y is the output vector. Initial conditions are preset to zero.

Parameters

- *Numerator coefficient*: The row vector of numerator coefficients.
- *Denominator coefficient*: The row vector of denominator coefficients.

4.6.5 Transport Delay

Description

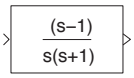
The Transport Delay block delays the input by a specified amount of time. It can be used to simulate a time delay. At the start of the simulation, the block outputs the *Initial output* parameter until the simulation time exceeds the *Time delay* parameter. Then the block begins generating the delayed input.

Parameters

- *Time delay*: The amount of simulation time that the input signal is delayed before being propagated to the output. The value must be nonnegative.
- *Initial output*: Specifies the output of the block until the simulation time exceeds the *Time delay* parameter.

4.6.6 Zero-Pole

Description



The Zero-Pole block models a system specified by the zeros, poles, and gain of a Laplace-domain transfer function that defines the relationship between the system's input and its outputs. This block assumes the following form for the transfer function:

$$h(s) = k\frac{(s-z_1)(s-z_2)\dots(s-z_{m-1})(s-z_m)}{(s-p_1)(s-p_2)\dots(s-p_{n-1})(s-p_n)}$$

where z_i represents the zeros, p_i the poles, and k the gain of the transfer function. The number of poles must be greater than or equal to the number of zeros. If the poles and zeros are complex, they must be complex conjugate pairs.

Parameters

- *Zeros*: The row vector of the zeros of the transfer function.
- *Poles*: The row vector of the poles of the transfer function.
- *Poles*: The gain of the transfer function.

4.7 Discontinuities

4.7.1 Saturation

Description

The Saturation block imposes upper and lower bounds on a signal. When the input signal is within the range specified by the *Lower limit* and *Upper limit* parameters, the input signal passes through unchanged. When the input signal is outside of these bounds, the signal is clipped to the upper or lower bound.

Parameters

- *Upper limit*: Specify the upper bound on the input signal. When the input signal to the Saturation block is above this value, the output of the block is clipped to this value.
- *Lower limit*: Specify the lower bound on the input signal. When the input signal to the Saturation block is below this value, the output of the block is clipped to this value.

4.8 Math Operations

4.8.1 Gain

Description

The Gain block multiplies the input by a constant value.

Parameters

- *Gain*: Specify the value by which to multiply the input. The gain may be a scalar, vector, or matrix.
- *Multiplication*: Specify the multiplication mode:
  `Element-wise(K*u)` – Each element of the input is multiplied by each element of the gain. The block performs expansions, if necessary, so that the input and gain have the same dimensions.
  `Matrix(K*u)` – The input and gain are matrix multiplied with the input as the second operand.
  `Matrix(u*K)` – The input and gain are matrix multiplied with the input

as the first operand.
`Matrix(K*u)(u vector)` – The input and gain are matrix multiplied with the input as the second operand. The input and the output are required to be vectors and their lengths are determined by the dimensions of the gain.

4.8.2 Math Function

Description

The Math Function block performs numerous common mathematical functions. The block output is the result of the operation of the function, specified by the *Function* parameter.

Parameters

- *Function*: Specify the mathematical function.

4.8.3 Product

Description

This block produces outputs using either element-wise or matrix multiplication, depending on the value of the *Multiplication* parameter. You specify the operations with the *Number of inputs* parameter. Multiply(*) and divide(/) characters indicate the operations to be performed on the inputs:

- If there are two or more inputs, then the number of characters must equal the number of inputs. For example, "*/*" requires three inputs. For this example, if the *Multiplication* parameter is set to Element-wise, the block divides the elements of the first input by the elements of the second input, and then multiplies the result by the elements of the third input. In this case, all nonscalar inputs to this block must have the same dimensions.
  If, however, the *Multiplication* parameter is set to Matrix, the block output is the matrix product of the inputs marked "*" and the inverse of inputs marked "/", with the order of operations following the entry in the *Number of inputs* parameter. The dimensions of the inputs must be such that the matrix product is defined.
- If only multiplication of inputs is required, then a numeric parameter value equal to the number of inputs can be supplied instead of "*" characters.

Parameters

- *Number of inputs*: Enter the number of inputs or a combination of "*" and "/" symbols.
- *Multiplication*: Specify element-wise or matrix multiplication.

4.8.4 Sum

Description

The Sum block performs addition or subtraction on its inputs. You specify the operations of the block with the *List of signs* parameter. Plus (+), minus (–), and spacer (—) characters indicate the operations to be performed on the inputs:

- If there are two or more inputs, then the number of "+" and "–" characters must equal the number of inputs. For example, "+ – +" requires three inputs and configures the block to subtract the second (middle) input from the first (top) input, and then add the third (bottom) input.
- A spacer character creates extra space between ports on the block's icon.
- If only addition of all inputs is required, then a numeric parameter value equal to the number of inputs can be supplied instead of "+" characters.

Parameters

- *Icon shape*: Designate the icon shape of the block.
- *List of signs*: Enter as many plus (+) and minus (–) characters as there are inputs. Addition is the default operation, so if you only want to add the inputs, enter the number of input ports.

4.9 Ports and Subsystems

4.9.1 Inport

Description

Inport blocks are the links from outside a system into the system. Simulink automatically assigns port numbers to Inport blocks.

Inport blocks in a subsystem represent inputs to the subsystem. A signal arriving at an input port on a Subsystem block flows out of the associated Inport block in that subsystem. The Inport block associated with an input

port on a Subsystem block is the block whose *Port number* parameter matches the relative position of the input port on the Subsystem block. For example, the Inport block whose *Port number* parameter is 1 gets its signal from the block connected to the topmost port on the Subsystem block. The Inport block name appears in the Subsystem icon as a port label. To suppress display of the label, select the Inport block and choose *Hide Name* from the *Format* menu.

Inport blocks in a top-level system can be used to supply external inputs from the workspace.

Parameters

- *Port number*: Specify the port number of the Inport block.

4.9.2 Outport

Description

Outport blocks are the links from a system to a destination outside the system. Simulink automatically assigns port numbers to Outport blocks.

Outport blocks in a subsystem represent outputs from the subsystem. A signal arriving at an Outport block in a subsystem flows out of the associated output port on that Subsystem block. The Outport block associated with an output port on a Subsystem block is the block whose *Port number* parameter matches the relative position of the output port on the Subsystem block. For example, the Outport block whose *Port number* parameter is 1 sends its signal to the block connected to the topmost output port on the Subsystem block. The Outport block name appears in the Subsystem icon as a port label. To suppress display of the label, select the Outport block and choose *Hide Name* from the *Format* menu.

Outport blocks in a top-level system can be used to supply outputs to the workspace.

Parameters

- *Port number*: Specify the port number of the Inport block.

4.9.3 Subsystem

Description

A Subsystem block represents a subsystem of the system that contains it. You can add blocks to the subsystem by opening the Subsystem block and copying blocks into its window.

The number of input ports drawn on the Subsystem block's icon corresponds to the number of Inport blocks in the subsystem. Similarly, the number of output ports drawn on the block corresponds to the number of Outport blocks in the subsystem.

4.10 Signal Routing

4.10.1 Demux

Description

The Demux block extracts the components of an input signal and outputs the components as separate signals. The output signals are ordered from top to bottom output port. The block accepts either vector signals or bus signals. The *Number of outputs* parameter allows you to specify the number and, optionally, the dimensionality of each output port. If you do not specify the dimensionality of the outputs, the block determines the dimensionality of the outputs for you.

The Demux block operates in either vector or bus selection mode, depending on whether you selected the *Bus selection mode* parameter. The two modes differ in the types of signals they accept. Vector mode accepts only a vector-like signal, that is, either a scalar or a column or row vector. Bus selection mode accepts only the output of a Mux block or another Demux block.

Parameters

- *Number of outputs*: The number and dimensions of outputs.

4.10.2 Mux

Description

The Mux block combines its inputs into a single vector output. An input can be a scalar or vector signal. The elements of the vector output signal take their order from the top to bottom, or left to right, input port signals. The Mux block's *Number of Inputs* parameter allows you to specify the number of inputs.

Parameters

- *Number of outputs*: The number and dimensions of inputs.

4.11 Sinks

4.11.1 Scope

Description

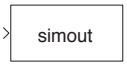
The Scope block displays its input with respect to simulation time. All axes have a common time range with independent y-axes. It easily lets you access any kind of signal during the simulation by double-clicking the block.

4.11.2 Terminator

Description

The Terminator block can be used to cap blocks whose output ports are not connected to other blocks. If you run a simulation with blocks having unconnected output ports, Simulink issues warning messages. Using Terminator blocks to cap those blocks avoids warning messages.

4.11.3 To Workspace



Description

The To Workspace block writes its input to the workspace. The block writes its output to an array or structure that has the name specified by the block's *Variable name* parameter. The *Save format* parameter determines the output format. It is recommended to use the format `array`. Selecting this option causes the To Workspace block to save the input as an N-dimensional array where N is one more than the number of dimensions of the input signal.

The way samples are stored in the array depends on whether the input signal is a scalar or vector or a matrix. If the input is a scalar or a vector, each input sample is output as a row of the array. For example, suppose that the name of the output array is `simout`. Then, `simout(1,:)` corresponds to the first sample, `simout(2,:)` corresponds to the second sample, etc.

Parameters

- *Variable name*: The name of the array that holds the data.

4.12 Example: Inverted Pendulum on a Cart [12]

The inverted pendulum described by its non-linear differential equations is simulated in series with an LQG controller. The observer gain matrix `L` and the optimal gain matrix `K` are built using the linearized state-space matrices. To investigate the influence of plant inaccuracy on the robustness of the system, the parameters of the linear state-space matrices, such as the length or the weight, differ slightly from those of the nonlinear plant.

The differential equations are given by:

$$\ddot{x} = \frac{F + ml(\dot{\theta}^2 \sin\theta - \ddot{\theta}\cos\theta)}{M + m}$$
$$\ddot{\theta} = \frac{g\sin\theta - \ddot{x}\cos\theta}{l}$$

The differential equations will be directly implemented in Simulink.

[12] for a sketch of the plant, see page 9

M-file

```
clc, clear all, close all

% Parameters:
M=1;                            % Weight of the cart [kg]
m=2;                            % Weight of the upper mass [kg]
l= 0.5;                         % Length of the pendulum [m]
g=9.81;                         % Gravitational constant [m/s^2]
tsim=10;                        % Simulation time [s]

% State-Space model:
A=[0 1 0 0; 58.86 0 0 0; 0 0 0 1; -19.62 0 0 0];
B=[0 -2 0 1]';
C=[1 0 0 0; 0 0 1 0];
D=[0 0]';

% LQG:
K=lqr(A,B,C'*C,1E-4);           % Optimal gain matrix
L=lqr(A',C',B*B',1E-5*eye(2))';  % Optimal observer matrix

% Set initial value of pendulum angle theta
theta_0=0.2;

% Simulate for different uncertainties
for i=1:0.1:1.3
    unc=i;
    M_real=M*unc;
    m_real=m*unc;
    l_real=l*unc;
    % Simulation of LQG_pend.mdl
    sim('LQG_pend',5)
    % Linear color interpolation (grayscales)
    figure(1)
    plot(tout,theta_out,'Color',[(i-1)/0.35,(i-1)/0.35,(i-1)/0.35]),hold on;
    figure(2)
    plot(tout,xout,'Color',[(i-1)/0.35,(i-1)/0.35,(i-1)/0.35]),hold on;
end
figure(1)
xlabel('time [s]');
ylabel('theta [rad]');
legend('U=0%','U=10%','U=20%','U=30%')
figure(2)
xlabel('time [s]');
ylabel('x_ cart [m]');
legend('U=0%','U=10%','U=20%','U=30%')
```

MDL-files

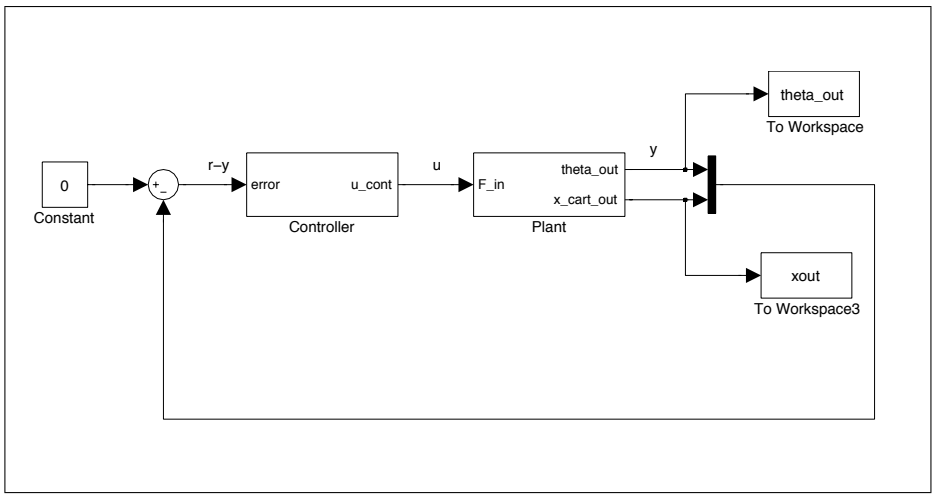


Fig. 4.3. Plant and Controller in Series

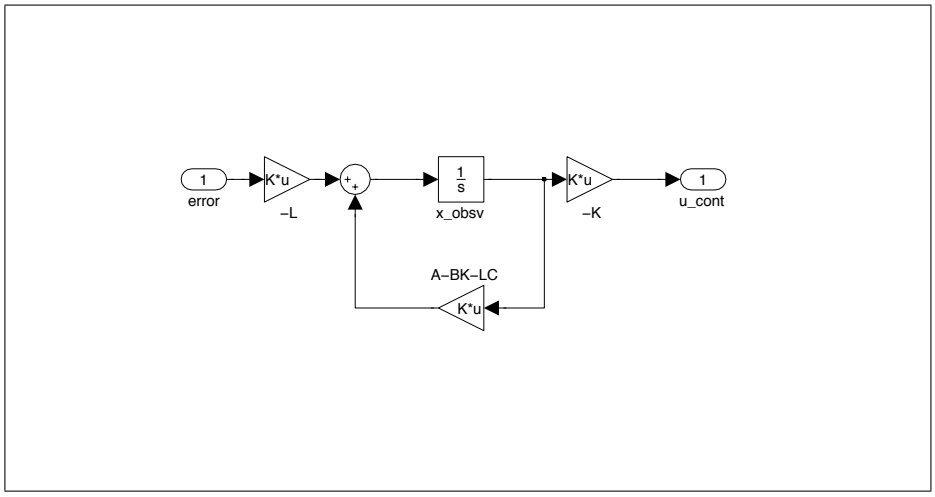


Fig. 4.4. LQG Controller

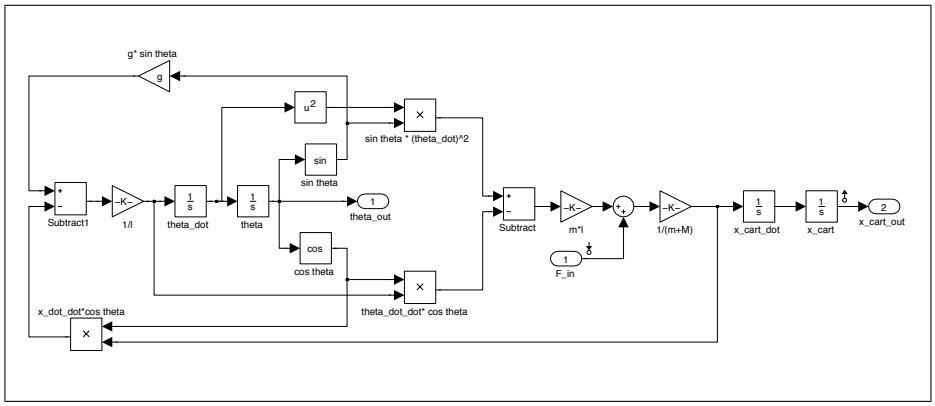


Fig. 4.5. Nonlinear Inverted Pendulum

Output

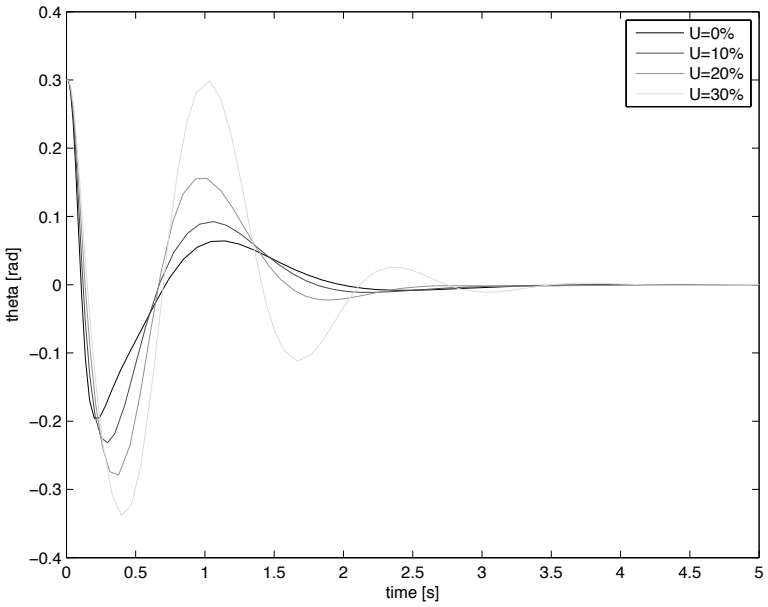


Fig. 4.6. Plot of theta versus time

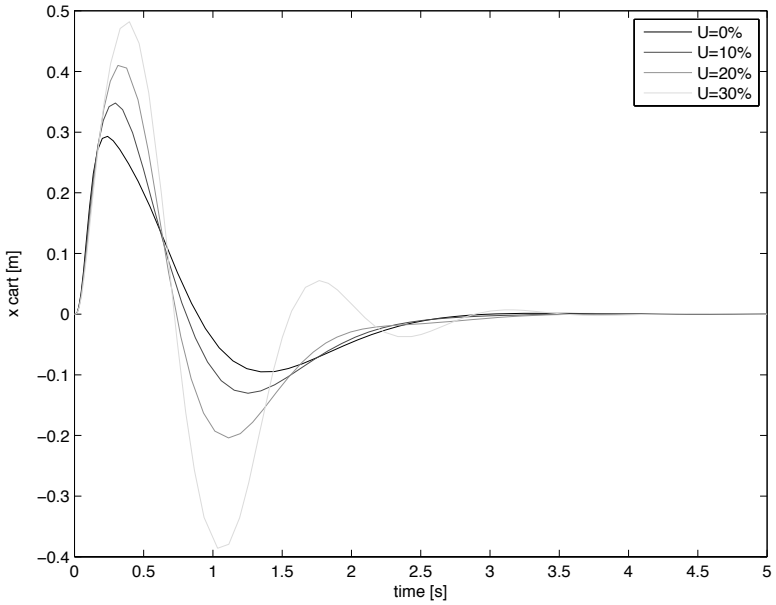


Fig. 4.7. Plot of the position of the cart versus time

Explanation

After creating the new model `LQG_pend.mdl`, we add two *Subsystem*[13] blocks. Later on, we will implement the controller and the plant inside them. Because we know that our plant has two outputs, we open the second subsystem and add another *Outport*[14] block.

The Top-Down approach simplifies the building of complex systems such that we can attend to each subsystem one at a time. In the main mdl-file, we add two *To Workspace*[15] blocks the export of the output variables `theta_out` and `xout`. We change the variable name by double-clicking the blocks. Using a *Mux*[16] block, we combine the output for feedback.

After opening the controller subsystem, we add the *Gain*[17], *Sum*[18] and *Integrator*[19] blocks. By double-clicking the *Integrator*, we can set an initial value. We leave it at zero. The parameters for the *Gain* blocks, however,

[13] see 4.9.3 Subsystem
[14] see 4.9.2 Outport
[15] see 4.11.3 To Workspace
[16] see 4.10.2 Mux
[17] see 4.8.1 Gain
[18] see 4.8.4 Sum
[19] see 4.6.2 Integrator

must be changed to *Matrix-multiplication*. Additionally, we set the gain to the corresponding variables in the M-file. The feedback arrow will be connected to the *Sum* block. For convenience, we change the names of all blocks. Always keep in mind that without proper documentation, you won't understand what you built after a few days.

Now, we are getting to the most difficult task in modeling a system in Simulink, the nonlinear model. A good start is to add all four states by adding *Integrator* blocks. To keep overview over the model, we change the names of the *Integrators* to the name of their respective output variables. The best thing to do now is to work at one differential equation at a time. Each equation should be formed in such a way that the highest derivative is on the left-hand side of the equation. Now, we can work our way inside-out the right-hand side of the equation: We start in the brackets, then add the blocks to build the fraction and so on. In the end, we connect the construct (containing the right hand side of the equation) to the input of the integrator of the highest derivative of the equation. After connecting all blocks, no empty output should occur.

The resulting block diagram of the overall system is depicted in Fig 4.3, the controller in Fig. 4.4 and the nonlinear system in Fig. 4.5.

For now, we are done working on the nonlinear subsystem, so we can start working on the M-file. We define all the variables we used while building our mdl-file. During the simulation, Simulink can access the MATLAB workspace to get the required variables. In the mdl-file, we utilize variables with the appendix `_real` to simulate the differences between the 'real' nonlinear model with the linearized controller. We also set the initial value of the pendulum's angle `theta` by defining the variable `theta_0`. We have to go back to our model, more specifically, to the *Integrator* `theta` in the nonlinear subsystem, and set the initial value to `theta_0`. We leave the initial state in the controller at zero. In reality, the initial states of the plant are never equal to those in the controller.

Finally, we come to the point of simulating the mdl-file. We call the function `sim(LQG_pend,5)`[20] to simulate the file `LQG_pend.mdl` for `5` seconds. We use a for-loop to investigate the effect of increasing model imprecisions (`unc`) on the performance of the system. As the resulting depicted in Fig. 4.7 shows, a higher overshoot results from these increased imprecisions.

[20] see 4.4 sim()