lektronik
abor

OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG
ELEKTRO- UND
INFORMATIONSTECHNIK

LTI and Cycle Based Modeling Using *Matlab*

Prof. Dr. Martin J. W. Schubert Electronics Laboratory
Ostbayerische Technische Hochschule Regensburg

Abstract. This tutorial explains how *Matlab* can be used to model linear and time-invariant (LTI) systems as (i) LTI systems in *s* or in *z* domain and (ii) as time-discrete finite-state machines (FSM).

1 Introduction

General

This communication explains how to translate system level behavioral models of *Simulink* and/or Spice into a finite state machine (FSM), which can then be translated into synthesizable code such as *VHDL* or *SystemC*. This step might be omitted in model based design (MBD), but it allows for deeper insight and control into the model than system-level design with *Simulink*. In addition, it can be performed with open source tools, e.g. *SystemC* or some free available *Matlab* clones like *Octave*, *Ocean* or *Scilab*.

This document assumes that we have modeled an RLC lowpass of a DC/DC buck converter as process in Laplace domain *s* and that PID control parameters are available. They may be optimized e.g. with *Matlab/Simulink*.

Modeling digital designs as LTI systems in *s* or *z* domain and/or as high-level cycle based models is an important tool of requirements engineering. Suitable tools as *Matlab* [1] or *System C* [2] allow for fast prototyping, before coding details using a hardware description language like *VHDL* [3] or *Verilog* [4] are implemented. The author prefers *Matlab* to *SystemC* for FSM design due to *Matlab's* powerful and easy to use LTI modeling and graphics capabilities.

Using Matlab

This document assumes some basic knowledge about *Matlab* and refers the reader to *David Houcque*'s *Introduction to Matlab for Engineering Students* [7].

Possibilities for processing selected the *Matlab* code lines in the following

- Select code lines and hit *F9* key.
- Define a cell between `%%` …. `%%` , position the cursor in it and run it with *CTRL+Return*.

The organization of this document is as follows:

Chapter 1 introduction,

Chapter 2 deals with LTI systems in *s* and *z* planes,

Chapter 3 explains how to translate an LTI model into a digital finite state machine (FSM),

Chapter 4 presents cycle-based methods to simulate FSMs with *Matlab* commands,

Chapter 5 draws relevant conclusion,

Chapter 6 offers some references,

Chapter 7 is an appendix offering some basics about PID controller design.

2 Linear and Time-Invariant (LTI) Systems

2.1 Mathematical Background

Linear and time invariant systems as defined e.g. in chapter 2 of the author's script about Linear Feedback Loops [8] can be shortly described as:

1. **A system is linear, when sinusoidal input causes a sinusoidal output with same frequency without adding offset or harmonics,**
2. **A system is time-invariant if its impulse response does not vary with time.**

2.1.1 Time-Continuous *LTI* Systems are Modeled in *s* (*Laplace*) Domain

A linear differential equation

$$b_0 y(t) + b_1 \frac{dy(t)}{dt} + \ldots + b_R \frac{d^R y(t)}{dt^R} = a_0 x(t) + a_1 \frac{dx(t)}{dt} + \ldots + a_R \frac{d^R x(t)}{dt^R} \tag{2.1.1}$$

Using $y(t) = \hat{Y} \cdot e^{s \cdot t}$ and $x(t) = \hat{X} \cdot e^{s \cdot t}$ delivers

$$\left[b_0 + b_1 s + \ldots + b_R s^R \right] \cdot y(t) = x(t) \cdot \left[a_0 + a_1 s + \ldots + a_R s^R \right] \tag{2.1.2}$$

This can be written in *Laplace* domain as transfer function

$$H(s) = \frac{Y(s)}{X(s)} = \frac{a_0 + a_1 s + \ldots + a_R s^R}{b_0 + b_1 s + \ldots + b_R s^R} \tag{2.1.3}$$

The frequency response of this system can be computed by setting $s = j\omega = j2\pi f$. (2.1.4)

In technical applications, particular attention is payed to second order systems that can be written in standard form as

$$H(s) = \frac{A_0}{\omega_0^2 + 2D\omega_0 s + s^2} \quad \text{or} \quad H(s) = \frac{A_0 (1 + s / s_{n1})}{\omega_0^2 + 2D\omega_0 s + s^2} \quad \text{or} \quad H(s) = \frac{A_0 (1 + s / s_{n1})(1 + s / s_{n2})}{\omega_0^2 + 2D\omega_0 s + s^2} \tag{2.1.5}$$

because they feature DC amplification A_0, damping parameter D poles and with radius $|s_p| = \omega_0$ when $0<D<1$. Systems with $D \geq 0$ are stable, systems with $D < 1$ oscillate and systems with $D < 0$ are unstable. One or two zeroes, s_{n1} and s_{n2}, respectively, may be added if required.

2.1.2 Translation from s → z

The accurate relationship between s and z is $z = e^{sT_s}$ with $T_s = 1/f_s$ being the sampling interval and f_s the sampling frequency.

Backward Euler method:

$$s = \frac{1 - z^{-1}}{T_s} \tag{2.2.1}$$

which stems from comparing derivatives $\frac{dx(t)}{dt} \cong \frac{x(n) - x(n-1)}{T_s} \Leftrightarrow s \cdot X \cong \frac{1 - z^{-1}}{T_s} X$

Forward Euler method:

$$s = \frac{z - 1}{T_s} \tag{2.2.2}$$

what is derived from comparing derivatives $\frac{dx(t)}{dt} \cong \frac{x(n+1) - x(n)}{T_s} \Leftrightarrow s \cdot X \cong \frac{z - 1}{T_s} X$

Bilinear (Tustin) method:

$$s = \frac{2}{T_s} \frac{1 - z^{-1}}{1 + z^{-1}} \tag{2.2.3}$$

what is derived from the mathematical series development

$$s = \frac{1}{T_s} \ln(z) = \frac{2}{T_s} \left[\frac{z-1}{z+1} + \frac{1}{3}\left(\frac{z-1}{z+1}\right)^3 + \frac{1}{5}\left(\frac{z-1}{z+1}\right)^5 + \ldots + \frac{1}{2n+1}\left(\frac{z-1}{z+1}\right)^{2n+1} + \ldots \right]$$

A symmetric back and forth translation delivers $s' = \frac{1 - z^{-1}}{1 + z^{-1}} \Leftrightarrow z^{-1} = \frac{1 - s'}{1 + s'}$ with $s' = s\frac{2}{T_s}$.

In this approximation $s = j\omega$ delivers $\left|z^{-1}\right| = 1 = \left|\frac{1 - j(\omega T_s / 2)}{1 + (\omega T_s / 2)}\right|$. Consequently, we have no amplitude error, but a compression of the frequency axis by *arctan*($\omega T_s/2$) in the z-domain.

2.1.3 Time-Discrete *LTI* Systems are Modeled in *z* Domain

Consider the time-domain equation for the time-discrete functions *x* and *y*:

$$c_0 y(n) + c_1 y(n-1) + c_2 y(n-2) = d_0 x(n) + d_1 x(n-1) + d_2 x(n-2) \tag{2.3.1}$$

with $x(n\text{-}k) = x(t_{n-k})$ sampled at equidistant time points $t_n = n \cdot T_s$, whereas with sampling frequency $f_s = 1/T_s$. For *LTI* functions we exploit the fact that delaying $x(t)$ by a sampling interval T_s corresponds to multiplying its spectrum $X(\mathrm{j}\omega)$ with $e^{-j\omega T_s}$, or multiplication by z^{-1} with $z=\mathrm{e}^{\mathrm{j}\omega T}$. We write $X(\mathrm{j}\omega) \rightarrow X(\mathrm{e}^{\mathrm{j}\omega T}) = X(\mathrm{z}(\mathrm{j}\omega)) = X(\mathrm{z})$. If $x(t)$ corresponds to $X(z)$, then $x(n\text{-}k)$ corresponds to $z^{-k} \cdot X(\mathrm{z})$. With *z* transform the time-domain equation in $x(n)$, $y(n)$ translates to

$$c_0 Y(z) + c_1 z^{-1} Y(z) + c_2 z^{-2} Y(z) = d_0 X(z) + d_1 z^{-1} X(z) + d_2 z^{-2} X(z) \tag{2.3.2}$$

Multiplication of z^{-1} corresponds to a time delay of sample period T_s. After factoring out $X(z)$ on the left and $Y(z)$ on the right hand side we write the transfer function

$$H(z) = \frac{Y(z)}{X(z)} = \frac{d_0 + d_1 z^{-1} + d_2 z^{-2}}{c_0 + c_1 z^{-1} + c_2 z^{-2}} = \frac{d_0 z^2 + d_1 z + d_2}{c_0 z^2 + c_1 z + c_2} \tag{2.3.3}$$

In signal processing applications we use z^{-1} rather than *z*, because multiplication with z^{-k} corresponds to $x(n\text{-}k)$, which is available as sampled in the past, while multiplication with z^k is sampled in the future, which is not available when causality is assumed.

Time-Domain Realization: Assume we multiply all coefficients of $H(z)$ with $c_0^{'} \neq 1$ to get

$$H(z) = \frac{d_0 + d_1 z^{-1} + d_2 z^{-2}}{1 + c_1 z^{-1} + c_2 z^{-2}} = \frac{d_0^{'} + d_1^{'} z^{-1} + d_1^{'} z^{-2}}{c_0^{'} + c_1^{'} z^{-1} + c_2^{'} z^{-2}}, \tag{2.3.4}$$

using coefficients $c_\#^{'}$, $d_\#^{'}$ (#=0,1,2) requires the evaluation of

$$y(n) = \frac{1}{c_0^{'}} \cdot \left(d_0^{'} x(n) + d_1^{'} x(n-1) + d_2^{'} x(n-2) - c_1^{'} y(n-1) - c_2^{'} y(n-2) \right) \tag{2.3.5}$$

with division by $c_0^{'}$ for computation of $y(n)$, n= 0, 1, 2, 3, …, causing computational effort. When working with floating point numbers we seek to get $c_0 = 1$, so that

$$y(n) = d_0 x(n) + d_1 x(n-1) + d_2 x(n-2) - c_1 y(n-1) - c_2 y(n-2). \tag{2.3.6}$$

When working with integral numbers and $c_0^{'} \neq 1$ is required, it is recommended to use $c_0^{'} = 2^M$ in Eq. (2.2.5) with $M \in \mathbb{N}$, because then division by $c_0^{'}$ can be realized simple and fast as bit-shift operation. Example: We need integer coefficients and have c_0=1, c_1=-1.7167, c_2=0.7167, d_0=0.6403, d_1=-1.2182, d_2=0.5793. Rounding would cause large errors. Accuracy improves considerably when we multiply all the coefficients e.g. by 2^{16} before rounding.

2.2 *Matlab's* LTI Systems with Data Type *sys*

Transfer functions *\$(s)*, *\$(z)* will be referred to as variables *TF\$_s*, *TF\$_z*, respectively, with *Matlab's* data type *sys*; for example *TFH_s*, *TFH_z* represent *H(s)*, *H(z)*, respectively.

2.2.1 Creating a Variable of *Matlab* Type *sys*

Type the commands of listing 2.2.1 into your *Matlab Command Window* one by one and observe the reactions of *Matlab*. Check for Matlab introduction in case of questions.

Listing 2.2.1: *Matlab* LTI system basics, time continuous

```
% File: L221_Matlab_LTI_System_Basics.m
clear all; % clear workspace
%
a0=0, a1=1; a2=2; % assign vaiables a0, a1, a2, ';' supresses echo
av=[a2 a1 a0]; % create vector av
bv=3:-1:1,     % create vector bv=[3 2 1] echoed in command window
%
% create a time-continuous transfer function with polynomials in s
TFH_s = tf(av, bv)
%
% create a time-continuous transfer function poles, zeros, contant in s
ZPK_s = zpk(av, bv, 2)
%
% translate ZPK_s to polynomial TF_ZPK_s
TFH_of_ZPK_s = tf(ZPK_s)
%
% translate TF_s to TF2ZPK_s
ZPK_of_TFH_s = zpk(TFH_s)
%
% a time-discrete system in z is created with sampling interval Ts
fs=1000; Ts=1/fs; % fs: sampling frequency fs=1KHz; Ts sampling interval Ts=1ms
TFH_z        = tf(av,bv,Ts)        % polynomial representation in z
ZPK_z        = zpk(av, bv, 2, Ts) % zero-pole-constatn representation in z
TFH_ZPK_z    = tf(ZPK_z)           % translate ZPK_z to TF_z
ZPK_of_TFH_z = zpk(TFH_z)          % translate TF_z to TF2ZPK_z
%
% Get back sys model paramters
[num_s,den_s] = tfdata(TFH_s,'v') % returns numerator and denominator (s)
[num_z,den_z] = tfdata(TFH_z,'v') % returns numerator and denominator (z)
```

Exercise: Complete the following *Matlab* statement such, that we get the echo printed below. (Note that *Matlab's* LTI systems note higher order coefficients first.)

```
» TFH_s =     tf([1 10], [1 4 6 0 1])
              .................................................

Transfer function:

        s + 10
------------------------
s^4 + 4 s^3 + 6 s^2 + 1
```

Complete the following *Matlab* statement such, that we get the Matlab echo printed below.

```
» TFH_z =     tf([1 10], [1 4 6 0 1],0.001)
              .................................................

Transfer function:

        z + 10
-----------------------------
z^4 + 4 z^3 + 6 z^2 + 1

Sampling time: 0.001
```

2.2.2 *Matlab's* Translation of LTI Systems from Continuous to Discrete

Listing 2.2.2 converts the model from time-continuous to time-discrete (*c2d*) domain and vice versa (*d2c*). Run listing 2.2.2 commands with listing 2.2.1 in the same directory.

Listing 2.2.2: Matlab LTI System Time Discretization. Run Listing 2.2.1 before

```
% File L222_LTI_Time_Discretization.m
clear all;                          % clear workspace
L221_Matlab_LTI_System_Basics       % run this m-file
fs=1e5, Ts=1/fs;                    % sampling rate and interval
TFH_s                               % display TF_s
TFH_z = c2d(TFH_s,Ts)               % s to z
TFH_s_from_TFH_z = d2c(TFH_z)       % z to s
```

2.2.3 Some Useful *Matlab* Statements

To *assert* stands for *make sur that*, *size* delivers number of row and columns of an array and statement *exist* checks for the existence of a name and stops execution if not. Try following codelines one by one. Note that you do not see "hello world!" if you uncomment the secons assert statement. Excape characters like '\n' are used as in *C* language.

Listing 2.2.3: Using some useful Matlab commands

```
% File L223_useful_commands.m
clear all;
help assert;        % get information about assert statement
help exist;         % get information about exist statement
help size;          % get information about size statement
a0 = 4;
e_a0=exist('a0'), e_A0=exist('A0') % Matlab is case sensitive!
A = ones(4,3),
sizeA = size(A)     % dimensions of A
rowsA = size(A,1)   % No of rows of A
colsA = size(A,2)   % No of columns of A
assert(4==4, '4 is not 5')  % Matlab carries on working
%assert(4==5, '4 is not 5') % Matlab stopps
fprintf('hello world!\n')   % print to command window
```

2.2.4 *Matlab* Graphics for LTI Systems

Listing 2.2.4: Matlab LTI Graphics. Run listings 2.1.1 and 2.1.3 before

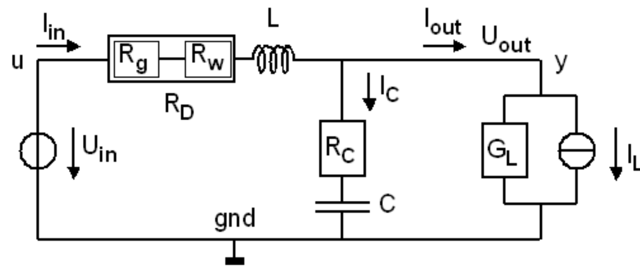
```
% File L224_LTI_Graphics.m
clear all;                                  % clear workspace
A0=10; D=sqrt(1/2); w0=100;                 % 2nd order model parameter
TFS_s=tf([A0*w0^2],[1 2*D*w0 w0^2])         % time-continuous TF
fs=100*w0; Ts=1/fs;                         % sampling interval
TFS_z=c2d(TFS_s,Ts)                         % time-discrete TF
figure(2241); bode(TFS_s,TFS_z); grid on;     % frequency response (Bode)
figure(2242); impulse(TFS_s,TFS_z); grid on;  % impulse response
figure(2243); step(TFS_s,TFS_z); grid on;     % step response
```

Listing 2.2.4 illustrates how to generate some graphics.

- Commands *impulse*(…) and delivers impulse responses of the lited LTI models.
- Commands *step*(…) and delivers step responses of the lited LTI models.
- Command *bode*(…) creates easily a Bode diagram but cannot be modified.

2.2.5 RLC Lowpass Model as LTI System Application

Fig. 2.2.5: RLC lowpass to model



Listing 2.2.5: File: *L225_LTI_RLC_lowpass_model.m*

```
% L225_LTI_RLC_lowpass_model
% ==========================
% Model of an RLC lowpass with load current IL and load resistor RL
% input paramters that mus be available in the workspace:
clear all;           % clear workspace
L=43e-6;             % L: serial inductance,
RD=0.2;              % RD: driver impedance = total resistance in series to L
C=680e-6;RC=0.05;    % C: vertical capacitance, RC: equivalent series resistor of C
RL=1e6; GL = 1/RL;   % RL: load impedance parallel to C+RC
%
% Process transfer function TFP models P(s)=Y(s)/U(s)
ap0=1;       ap1=C*RC;                    ap2=0;
bp0=1+RD*GL; bp1=(RC+RD+RD*RC*GL)*C+GL*L; bp2=(1+RC*GL)*L*C;
TFP_s = tf([ap2 ap1 ap0], [bp2 bp1 bp0]); % TFP_s is a Matlab LTI system
% Inference transfer function TFQ mdels Q(S)=Y(s)/U(s)
aq0=-RD; aq1=-(RC*RD*C+L); aq2=-RC*L*C; bq0=bp0; bq1=bp1; bq2=bp2;
TFQ_s = tf([aq2 aq1 aq0], [bq2 bq1 bq0]); % TFQ_s is a Matlab LTI system
% Graphical Postprocessing
figure(225);
bop=bodeoptions(TFP_s);
bop.FreqUnits='Hz';bop.Title.FontSize=11; bp.Title.FontWeight='bold';
bop.Title.String='Bode Diagram: U_o_u_t / U_i_n';
subplot(221); bodeplot(TFP_s,'b',bop); grid on;
subplot(222); step(TFP_s,'b');      grid on;
boq=bop; bop.Title.String='Bode Diagram: U_o_u_t / I_L_o_a_d';
subplot(223); bodeplot(TFQ_s,'r',boq); grid on;
subplot(224); step(TFQ_s,'r');      grid on;
%
%% Equivalent 2nd order model with standard parameter A0, D, w0
A0  = 1/(1+RD*GL);            % DC amplification of the configuration
DC  = 0.5*(RD+RC+RD*RC*GL) / ((1+RD*GL)*(1+RC*GL)) * sqrt(C/L);
DL  = 0.5*        GL        / ((1+RD*GL)*(1+RC*GL)) * sqrt(L/C);
D   = DC + DL;                % damping constant
w0  = sqrt((1+RD*GL)/((1+RC*GL)*L*C)); % asymptotes intercept freq.
wn1 = 1/(RC*C);               % zero caused by capacitor's ESR
wn2 = 1/(GL*L);               % infinite: no 2nd zero in transfer function
wp2 = w0*(D+sqrt(D*D-1));     % higher pole frequency in Hz
wp1 = w0*(D-sqrt(D*D-1));     % lower  pole frequency in Hz
%
% corresponding frequencies in Hz:
pi2=pi*2; f0=w0/pi2; fn1=wn1/pi2; fn2=wn2/pi2; fp1=wp1/pi2; fp2=wp2/pi2;
fprintf('\nf0 : intercept of 0dB/-40dB asymptotes: %d',f0);
fprintf('\nfn1: zero for STF and QTF: %d',fn1);
fprintf('\nfn2: zero for QPF only: %d',fn2);
fprintf('\nfp1: 1st pole: %d, angle: %d°',abs(fp1),angle(fp1)*180/pi);
fprintf('\nfp2: 2nd pole: %d, angle: %d°',abs(fp2),angle(fp2)*180/pi);
fprintf('\n');
```

Listing 2.2.5 models the RLC lowpass of Fig. 2.4 and plots its characteristics. Command *bodeplot* is by default identical to *bode* but allows setting options as demonstrated with *Bode* options variable *bop* in listing 2.2.5. Command *fprintf* prints into the Command window.

2.3 Self-Defined LTI Systems Data Structure

Goal: Describe LTI systems independently of *Matlab's* data type *sys*, as it is hardly portable to other programming languages or hardware description languages such as *SystemC* or *VHDL*, respectively.

Transfer functions *$*(*s*), *$*(*z*) will be referred to as variables *$TF_s*, *$TF_z*, respectively, with *Matlab's* data type *sys*; for example *HTF_s*, *HTF_z* represent *H(s)*, *H*(*z*), respectively.

Remember that LTI systems in *Matlab* notation were named *TF$_s* and *TF$_z* for *$*(*s*) and *$*(*z*), respectively.

LTI systems defined here are vectors named *$TF_s* and *$TF_z* for *$*(*s*) and *$*(*z*), respectively, having a vector length of *R*+3 to 2*R*+4, whereas missing trailing vector elements default to 0 with exception of element *R*+4, which defaults to 1.

(a) Time-continuous system data object

Time-continuous System:	$STF(s)=\frac{Y(s)}{X(s)}=\frac{a_0+a_1s+...+a_Rs^R}{b_0+b_1s+...+b_Rs^R}$							
Order *R*:	Contents of time- continuous LTI system object *$TF*(*s*) of order *R*							
Index #	1	2	3	…	*R*+3	*R*+4	…	2*R*+4
$TF_s(#)	*0*	*R*	a_0	...	a_R	b_0	...	b_R
Order *2*:	Contents of time-continuous LTI system object *$TF*(*s*) of order 2							
Index #	1	2	3	4	5	6	7	8
$TF_z(#)	*0*	*2*	a_0	a_1	a_2	b_2	b_1	b_2

(b) Time-discrete system data object, c_0 ... c_R may be omitted and default to 1 0…0

Time-discrete System:	$STF(z)=\frac{Y(z)}{X(z)}=\frac{d_0+d_1z^{-1}+...+d_Rz^{-R}}{c_0+c_1z^{-1}+...+c_Rz^{-R}}$							
Order *R*:	Contents of time-discrete LTI system object *$TF*(*z*) of order *R*							
index	1	2	3	…	*R*+3	*R*+4	…	2*R*+4
content	T_s	*R*	d_0	...	d_R	c_0	...	c_R
Order *2*:	Contents of time-discrete LTI system object *$TF*(*z*) of order 2							
index	1	2	3	4	5	6	7	8
content	T_s	*2*	d_0	d_1	d_2	c_0	c_1	c_2

Table 2.3.1: Notation of self-defined LTI systems with order *R* and sampling interval T_s.

The polynomial notation $STF(s) = a_0 + a_1\cdot s + a_2\cdot s^2 + \dots$ is different to Matlab LTI system notation, where the highest coefficients are noted first. Therefore, we use function *f_h* to flip vectors or even matrixes horizontally. Horizontally flipping is required for LTI systems in *s* but not for LTI systems in *z*, because Matlab uses notation in *z* while our self-defined models use notation in z^{-1}, whereas multiplication with *z* and z^{-1} models future and past, respectively.

Listing 2.3.1: Matlab code to flip matrix *M* horizontal

```
% File f_h.m: flip horizontal: matrix M
function M_flipped_horizontal=f_h(M)
ncols=size(M,2);
for col=1:ncols;
  M_flipped_horizontal(:,col) = M(:,ncols+1-col);
end;
```

Function *f_c2d_bilin_order2* performs translation of a second order transfer function from $s \rightarrow z$ using bilinear (Tustin) substitution according to eq. (3.2) similar like Matlab's *c2d* command. (A general order function *f_c2d* would be difficult to read.)

Listing 2.3.2: *Matlab* code for bilinear (*Tustin*) transformation $s \rightarrow z$ of a biquad model

```
%%%%%%%%%1%%%%%%%%%2%%%%%%%%%3%%%%%%%%%4%%%%%%%%%5%%%%%%%%%6%%%%%%%%%7%%%%%
%
% Function: f_c2d_bilin_order2: continuous to disctrete
%
% Purpose: translate 2nd oRder (R=2) transfer function (TF) coefficients
%          from continuous to discrete with bilinear (Tustin) transform
%          s = (2/Ts)*(1-z^-1)/(1+z^-1).
%
% Output arguments: STF_z == [Ts R d0 d1 d2 c0 c1 c2];
%                            [ 1 c1 c2]]
% Input arguments:
%  required: STF_s == [0 R a0 a1 a2 b0 b1 b2];
%              Ts: = 1/fs with sampling frequency fs. Default: fs=1e6
%
% Function Calls: <none>
% Coded by: Martin Schubert, date of last change: 14.Jan.2018
%
%%%%%%%%%1%%%%%%%%%2%%%%%%%%%3%%%%%%%%%4%%%%%%%%%5%%%%%%%%%6%%%%%%%%%7%%%%
%
function STF_z = f_c2d_bilin_order2(STF_s,Ts)
%
fs2=2/Ts;
a0=STF_s(3); a1=STF_s(4); a2=STF_s(5);
b0=STF_s(6); b1=STF_s(7); b2=STF_s(8);
as0=a0; as1=a1*fs2; as2=a2*fs2*fs2;
bs0=b0; bs1=b1*fs2; bs2=b2*fs2*fs2;
cs0=bs0+bs1+bs2; c1=2*(bs0-bs2)/cs0; c2=(bs0-bs1+bs2)/cs0;
d0=(as0+as1+as2)/cs0; d1=2*(as0-as2)/cs0; d2=(as0-as1+as2)/cs0;
STF_z = [Ts 2 d0 d1 d2 1 c1 c2];
```

Listing 2.3.3: Translation to data type *sys* to use *Matlab's* LTI graphics commands

```
% L233_LTI_System_translation_selfmade2matlab.m
clear all;
A0 = 10; D=1/2; f0=1e4; w0=2*pi*f0;
a0=A0*w0^2; a1=0,      a2=0;
b0=w0^2;    b1=2*D*w0; b2=1;
fs=1e5; Ts=1/fs;                      % sampling rate and period, resp.
HTF_s = [0, 2, a0 a1 a2, b0 b1 b2]    % selfmade TF matrix model in s-domain
HTF_z = f_c2d_bilin_order2(HTF_s,Ts) % selfmade TF matrix model in z-domain
% translate to Matlab type sys to use Matlab graphics
TFH_s = tf(f_h(HTF_s(3:5)), f_h(HTF_s(6:8)))
TFH_z = tf(HTF_z(3:5),HTF_z(6:8),Ts)
figure(2331); step(TFH_s,TFH_z);
bop=bodeoptions(TFH_s); bop.FreqUnits='Hz'; bop.Ylim=[-200,30];
figure(2332); bodeplot(TFH_s,'b',TFH_z,'r',bop); grid on;
```

Listing 2.3.4: read numerator and denominator from a *Matlab* variable with type sys.

```
[num,den]=tfdata(TFH,'v') % return numerator and denominator of TFH with type sys
```

2.4 Exercises:

Given is the 2nd order LTI system $H(s) = \dfrac{A_0}{\omega_0^2 + 2D\omega_0 s + s^2}$. Complete the code below:

```
% L24_LTI_Systems_exercise.m
% *************************
%  clear workspace
%
clear all;                                      % clear workspace
% ......................................................
%
A0=10; D=0.1; f0=100; w0=2*pi*f0; % 2nd order standard model parameters
%
% Working with variables of Matlab's data type sys
% ================================================
% Create Matlab type sys variable with standard parameters above
%
TFS_s = tf([A0*w0^2],[1 2*D*w0 w0^2])           % time-continuous TF
%      ................................................
% Compute sampling interval Ts from sampling frequency fs
%
fs=10*f0; Ts =  1/fs;                           % sampling interval
%            .........................................
%
%  Translate system TFS_s into a time-discrete system TFS_z using Ts
%
TFS_z = c2d(TFS_s,Ts)                           % time-discrete TF
%
% Plot Bode diagram of TFS_s and TFS_z into figure with handle 2241
%
figure(221);  bode(TFS_s,TFS_z);                % frequency response
%            .........................................
%
% Plot implse responses of TFS_s and TFS_z into figure with handle 222
%
figure(222); impulse(TFS_s,TFS_z);               % impulse response
% .......................................................
%
% Plot step responses of TFS_s and TFS_z into figure 223 with grid on
%
figure(223); step(TFS_s,TFS_z); grid on;          % step response
% ........................................................% with grid on
%
% Translate Matlab data type sys variable TFS_z to selfmade variable STF_s
% ========================================================================
% compute numerator and denominator of z from TFS_z
%
[num_z,den_z] = tfdata(TFS_z,'v')
% ........................................................
%
% Create from numZ, den_z the 2nd order LTI system STF_z in self-defined format
%
STF_z = [Ts, 2, num_z, den_z]
% ........................................................
%
% compute numerator and denominator of s from TFS_s
%
[num_s,den_s] = tfdata(TFS_s,'v')
% ........................................................
%
% Create 2nd order time-discrete LTI system STF_z in self-defined format
%
STF_s = [0, 2, f_h(num_s), f_h(den_s)]
% ........................................................
```

Given is $H(s) = \dfrac{Y(s)}{X(s)} = \dfrac{a_0 + a_1 s + ... + a_R s^R}{b_0 + b_1 s + ... + b_R s^R}$

Write the corresponding time-domain differential equation.

$$b_0 y(t) + b_1 \dot{y}(t) + b_2 \ddot{y}(t) = a_0 x(t) + a_1 \dot{x}(t) + a_2 \ddot{x}(t)$$

. . . .

Given is $H(z) = \dfrac{Y(z)}{X(z)} = \dfrac{d_0 + d_1 z^{-1} + d_2 z^{-2}}{c_0 + c_1 z^{-1} + c_2 z^{-2}}$

Write the corresponding time-domain differences equation.

$$c_0 y_n + c_1 y_{n-1} + c_2 y_{n-2} = d_0 x_n + d_1 x_{n-1} + d_2 x_{n-2}$$

. . . .

Do at least some of the following translations (with paper and pencil)

Use Substitution $s = f_s(1 - z^{-1})$ and $a_k^* = a_k \left(f_s\right)^k$, $b_k^* = b_k \left(f_s\right)^k$, $k = 0,1,2$.

Given:	**Wanted::**	**Calculate**
$H_0(s) = \dfrac{a_0}{b_0}$	$H_0(z^{-1}) =$	$\dfrac{a_0}{b_0}$
$H_1(s) = \dfrac{a_0 + a_1 s}{b_0 + b_1 s}$	$H_1(z^{-1}) =$	$\dfrac{(a_0^* + a_1^*) + (-a_1^*) z^{-1}}{(b_0^* + b_1^*) + (-b_1^*) z^{-1}}$
$H_2(s) = \dfrac{a_0 + a_1 s + a_2 s^2}{b_0 + b_1 s + b_2 s^2}$	$H_2(z^{-1}) =$	$\dfrac{(a_0^* + a_1^* + a_2^*) + (-a_1^* - 2a_2^*) z^{-1} + a_2^* z^{-2}}{(b_0^* + b_1^* + b_2^*) + (-b_1^* - 2b_2^*) z^{-1} + b_2^* z^{-2}}$

Use Substitution $s = 2 f_s \dfrac{1 - z^{-1}}{1 + z^{-1}}$ and $a_k^* = a_k \left(2 f_s\right)^k$, $b_k^* = b_k \left(2 f_s\right)^k$, $k = 0,1,2$.

Given:	**Wanted::**	**Calculate**
$H_0(s) = \dfrac{a_0}{b_0}$	$H_0(z^{-1}) =$	$\dfrac{a_0}{b_0}$
$H_1(s) = \dfrac{a_0 + a_1 s}{b_0 + b_1 s}$	$H_1(z^{-1}) =$	$\dfrac{(a_0^* + a_1^*) + (a_0^* - a_1^*) z^{-1}}{(b_0^* + b_1^*) + (b_0^* - b_1^*) z^{-1}}$
$H_2(s) = \dfrac{a_0 + a_1 s + a_2 s^2}{b_0 + b_1 s + b_2 s^2}$	$H_2(z^{-1}) =$	$\dfrac{(a_0^* + a_1^* + a_2^*) + 2(a_0^* - 2a_2^*) z^{-1} + (a_0^* - a_1^* + a_2^*) z^{-2}}{(b_0^* + b_1^* + b_2^*) + 2(b_0^* - 2b_2^*) z^{-1} + (b_0^* + b_1^* + b_2^*) z^{-2}}$

Compare your result to Matlab function *f_c2d_bilin_order2*.

3 Translating an LTI Model in *z* to a Finite State Machine

3.1 Translating LTI Models in *z* to Finite Differences Equations

Given is a function $H(z)$ to be realized as finite state machine (FSM). If the transfer function is given in s, translate it to z first. Transfer functions can be written in both z and z^{-1}:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{d_0 z^2 + d_1 z^1 + d_2}{c_0 z^2 + c_1 z + c_2 z} = \frac{d_0 + d_1 z^{-1} + d_2 z^{-2}}{c_0 + c_1 z^{-1} + c_2 z^{-2}} \tag{3.1}$$

with frequency domain input and output functions X and Y, respectively. Control engineers typically use models in z. This requires order of denominator ≥ order of the numerator, as multiplication with z is using a sample in the future. Signal processing engineers often prefer models in z^{-1}, as samples in the past remain available when saved in memory. Eq. (3.1) can be written as

$$Y(z)\left(c_0 + c_1 z^{-1} + c_2 z^{-2}\right) = X(z)\left(d_0 + d_1 z^{-1} + d_2 z^{-2}\right). \tag{3.2}$$

and consequently as

$$c_0 Y(z) + c_1 z^{-1} Y(z) + c_2 z^{-2} Y(z) = d_0 X(z) + d_1 z^{-1} X(z) + d_2 z^{-2} X(z) \tag{3.3}$$

Transferring (3.3) into time domain yields

$$c_0 y_n + c_1 y_{n-1} + c_2 y_{n-2} = d_0 x_n + d_1 x_{n-1} + d_2 x_{n-2} \tag{3.4}$$

with x_{n-k}, y_{n-k} being input and output samples, respectively, at time $t_{n-k}, = (n-k)\cdot T_s$ with n, k being integral numbers and $T_s = 1/f_s$ the sampling interval. At sample time t_n we compute output signal y_n as

$$y_n = \frac{1}{c_0}\left[d_0 x_n + d_1 x_{n-1} + d_2 x_{n-2} - c_1 y_{n-1} - c_2 y_{n-2}\right] \tag{3.5}$$

with x_{n-k}, y_{n-k} being samples that are k clock periods older than x_n, y_n, respectively.

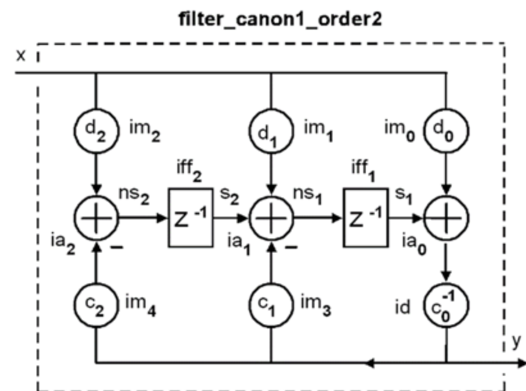


Fig. 3.1: Time-discrete filter in 1st canonical direct structure.

Fig. 3.1 illustrates a 2nd order filter in 1st canonical direct structure. It is direct, because the impulse response taps can be directly defined with the coefficients. (The same result can also be achieved with a filter in a 2nd canonical direct structure, which will not be discussed here.)

If possible, the time consuming division by c_0 is avoided by dividing the whole equation by c_0. This is the typical case for theoretical analytics.

For digital synthesis, when all coefficients must be integral numbers or bit-vectors, we may need $c_0 > 1$ for accuracy reasons. Then we should use $c_0 = 2^M$, for example 2^{16}, because this reduces the division by c_0 to a simple and fast M bit shift-right operation.

In eq. (3.5), we have to memorize 4 values from the past for a 2nd order model, namely x_{n-1}, x_{n-2}, y_{n-1} and y_{n-2}. In general, for a model of order R we need $2R$ memory elements. In signal processing there is a method to realize LTI systems of order R with R memory elements only, which are then said to canonical.

3.2 Translating Difference Equation to a Finite State Machine

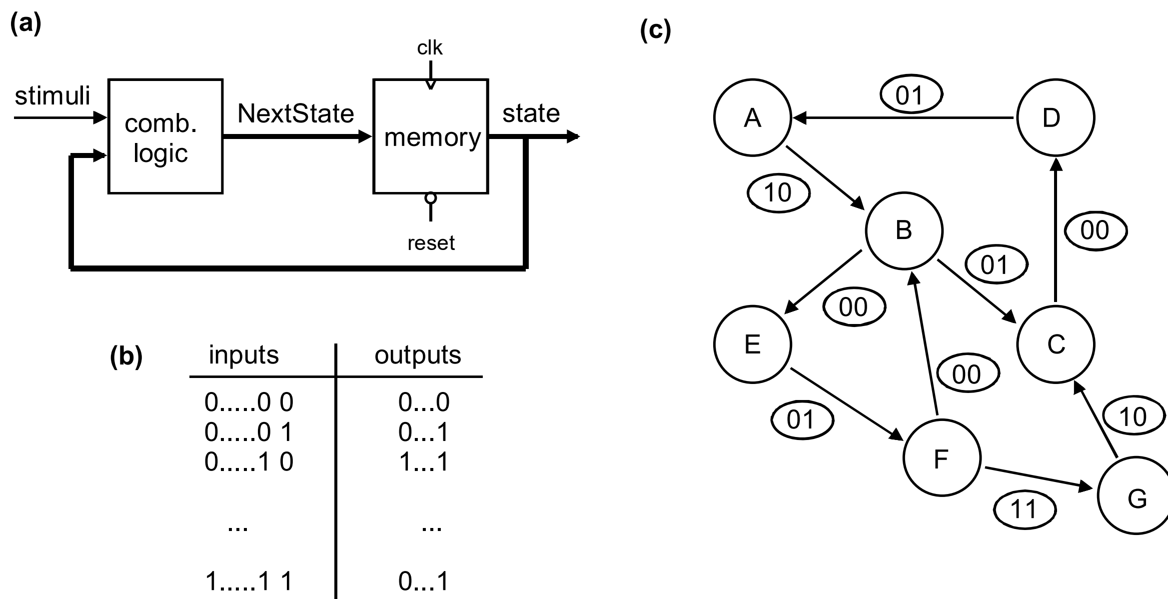


Fig. 3.2: different finite-state machine (FSM) models: **(a)** loop, **(b)** table, **(c)** bubbles.

Cycle-base modeling is oriented to the loop model of Fig. 3.2(a). Memory is a variable and *nextstate* logic a function. In the loop, the *nextstate* logic computes the *nextstate* from *state* and *stimuli* and assigns it to *state*.

Memory, which is drawn as boxes labeled z^{-1} in Fig. 3.1, is typically realized with flipflops. It contains the complete relevant information of the past. In- and output signals of memory have the same data type and are generally termed *nextstate* and *state*, and in Fig. 3.1 they are named *ns* and *s*, respectively.

***Nextstate* logic** contains combinational logic only, i.e. it is free of memory. Thus, it delivers (after a short settling time) a static signal named *nextstate*, which is a function of the actual *state* and *stimuli* only, but neither on the past nor on itself.

Cycle-based models are significantly faster to design and simulations run some 300…1000 times faster than the following evet-driven models (e.g. with VHDL), that are the last stage before synthesis and download into an FPGA.

3.3 FSM Model for Digital Circuit Design

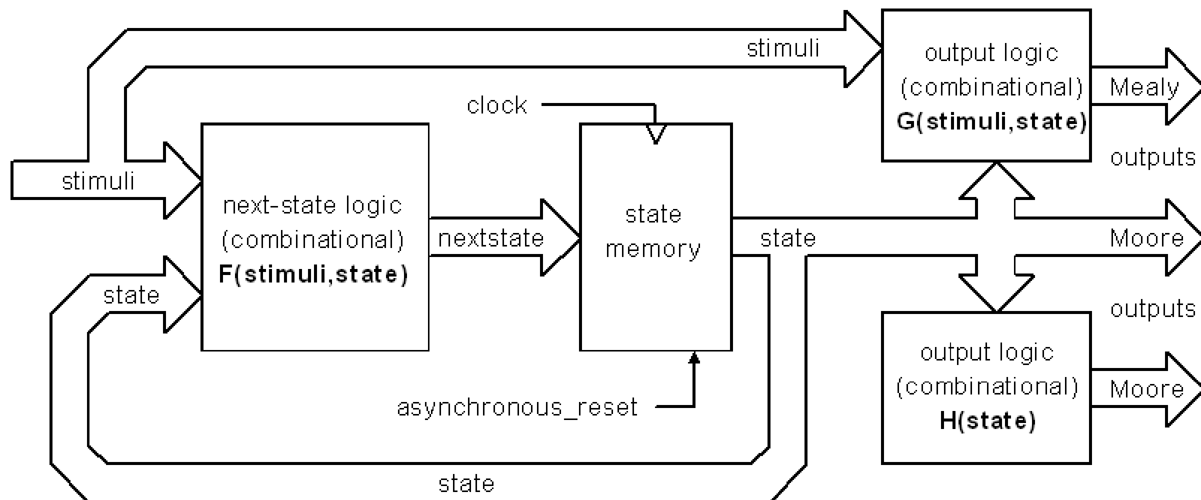


Fig. 3.3: General finite state machine (FSM) model suitable for digital circuit design.

Fig. 3.3 illustrates the general loop-based finite state machine model. The outputs are distinguished between *Moore* outputs and *Mealy* outputs, depending whether they depend on the *state* only or not, respectively. Wherever possible we should try to design Moore outputs, because they are reliably constant until the next active clock edge.

The state of a finite state machine (FSM) is finite, because its *N* flipflops can represent a limited (finite) number of 2^N states, with *N* being integral. An example for an infinite state machine is an RC lowpass: Its capacitor can hold an unlimited (infinite) number of voltages.

In real designs, memory changes *state* only in response to the active clock edge (after all *nextstate* signals have settled to static values) to take over *nextstate* as *state*. Only exception is an asynchronous global *reset* at time zero, which corresponds to the initial *state* in cycle-based modeling.

For the rest of this communication we will train cycle-based FSM modeling and simulation starting with a simple counter.

3.4 Exercises to Chapter 3

Given is

$$H(z)=\frac{Y(z)}{X(z)}=\frac{d_0+d_1z^{-1}+d_2z^{-2}}{c_0+c_1z^{-1}+c_2z^{-2}}$$

Write the corresponding time-domain differences equation.

$$c_0y_n+c_1y_{n-1}+c_2y_{n-2}=d_0x_n+d_1x_{n-1}+d_2x_{n-2}$$

. . . .

Write the corresponding time-domain differences equation $y_n = f(x_{n-k}, y_{n-k})$, k=0,1,2

y_n = $\frac{1}{c_0}\left[d_0x_n+d_1x_{n-1}+d_2x_{n-2}-c_1y_{n-1}-c_2y_{n-2}\right]$

. . . .

To avoid unnecessary computing power, how do we modify this equation when working with floating-point numbers as coefficients?

Multiply all coefficients by (1/c_0) getting $c_{0,new}$ = 1

. . . .

. . . .

To avoid unnecessary computing power, how do we modify this equation when working with integer numbers as coefficients, where other coefficients by be close to or less than 1?

Multiply all coefficients by $2^M/c_0$ → $c_{0,new}$ = 2^M, M integral

. . . .

. . . .

Why does this means save computing power?

Division by $c_{0,new}$=2^M, M integral, is a simple M bit-shift right

. . . .

. . . .

4 Cycle-Based FSM Modeling with *Matlab*

4.1 *Matlab* Counter Model as Finite State Machine (FSM)

4.1.1 Evaluate FSM Model

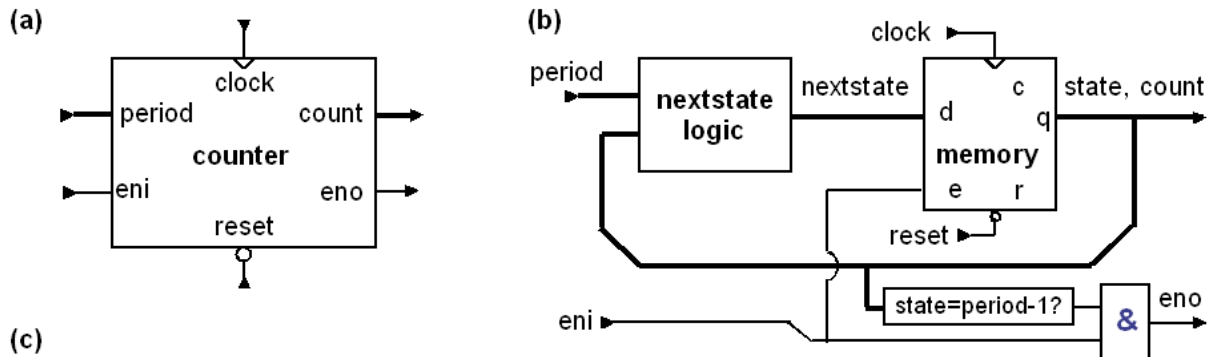


Fig. 4.1.1: Counter: **(a)** symbol, **(b)** schematics.

Exercise: Assign signal names in Fig. 4.1.1(b) to signals of Fig. 3.3

Stimuli: period, eni

Nextstate: nextstate

State: state

Nextstate Logic: if eni == 1 then

{nextstate = state+1 when state < period-1, else nextstate = 0}

Moore Outputs: count

Mealy Outputs: eno

Output Logic Moore: count = state

Output Logic Mealy: eno = (state == period-1) & eni

Which signals have to be initialized like? state @ t=0,

all stimuli (period, eni) for any t

4.1.2 Realize Combinational Logic of the Counter as Function

The *Matlab* code lines copied from testbench *tb_countcl*, listing 4.1.2.1

```
s=0; % initialize state
for n=1:NoS;
  % evaluate counte combinational logic:
  [y(n),eno(n),ns] =
f_countcl(period,eni(n),s);
  % apply active clock edge:
  s = ns; % state = nextstate
end; % time loop over time index n
```

use function *f_countcl* for combinational logic only. The state memory "**s=ns**" is realized outside function *f_countcl*.

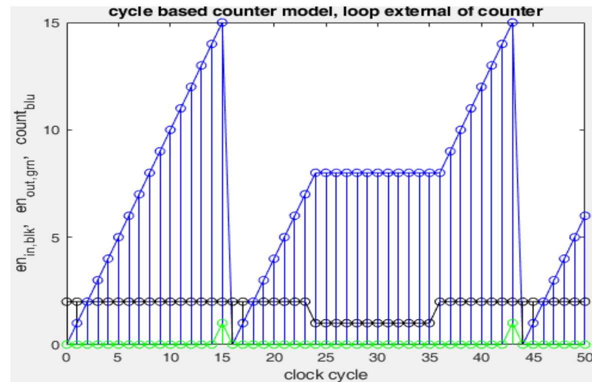


Fig. 4.1.2 as generated by testbench *tb_countcl* shown in listing 4.1.1.

Listing 4.1.2.1: Testbench *tb_countcl.m* uses function *f_countcl* as nextstate logic

```
% Testbench tb_countcl.m
% initializations
NoS    = 51;  % Number of Samples
period = 16;                      % stimuli: max{count}
eni = logical(ones(1,NoS));       % stimuli: input enable
eni(25:36) = logical(zeros(1,12)); % stimuli: input enable
%
% 1. work off the FSM sample by sample
s=0; % initialize state
for n=1:NoS;
  % evaluate the counter's combinational logic:
  [y(n),eno(n),ns] = f_countcl(period,eni(n),s);
  % apply active clock edge:
  s = ns; % state = nextstate
end; % time loop over time index n
%
% Graphical Postprocessing
figure(421);
t=0:NoS-1;
subplot(111);
plot(t,eni+1,'k',t,y,'b',t,eno,'g'); hold on;
stem(t,eni+1,'k');stem(t,y,'b');stem(t,eno,'g'); hold off;
title('cycle based counter model, loop external of counter')
xlabel('clock cycle');
ylabel('en_i_n_,_b_l_k,   en_o_u_t_,_g_r_n,   count_b_l_u');
```

Listing 4.1.2.2: function *f_countcl* contains nextstate logic only

```
function [count,eno,nextstate]=f_countcl(period,eni,state)
% initialize state if not available
if exist('state')~=1; state=0; end;
%
  % NextState Logic
  if eni==0;
    nextstate = state;
  else
    if state < period-1;
      nextstate = state+1;
    else
      nextstate = 0;
    end;
  end;
  %
  %Output-Logic:
  count = state;                  % Moore output
  eno = (state == period-1) & eni; % Mealy output
```

4.1.3 Realize Complete Counter as a Function

The *Matlab* code lines

```
s=0; % initialize state
for n=1:NoS;
   [y1(n),eno1(n),ns] = …
      f_counter(period,eni(n),s);
   % apply active clock edge:
   s = ns; % state = nextstate
end; % time loop over time index n
%
% 2. realize complete time axis
within function f_counter
[y2,eno2] = f_counter(period,eni);
```

of testbench *tb_counter.m* in listing 4.1.3.1 use function *f_counter* (i) as combinational logic only to compute *f* and (ii) as complete FSM to compute *y*.

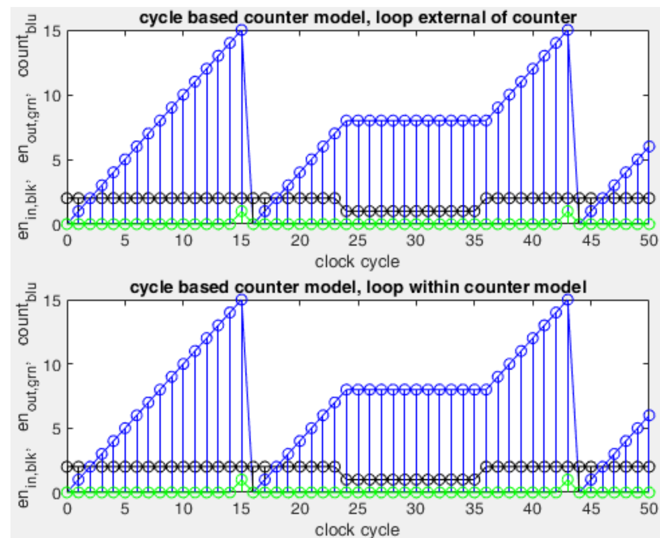


Fig. 4.1.3 as generated by testbench *tb_counter* shown in listing 4.2.1.

Listing 4.1.3.1: Testbench *tb_counter.m* uses function *f_counter* as nextstate logic

```
% L431: Testbench tb_counter.m
% initializations
NoS    = 51;  % Number of Samples
period = 16;                       % stimuli: max{count}
eni = logical(ones(1,NoS));        % stimuli: input enable
eni(25:36) = logical(zeros(1,12)); % stimuli: input enable
%
% 1. index n loops over time axis sample by sample
s=0; % initialize state
for n=1:NoS;
  [y1(n),eno1(n),ns] = f_counter(period,eni(n),s);
  % apply active clock edge:
  s = ns; % state = nextstate
end; % time loop over time index n
%
% 2. realize complete time axis within function f_counter
[y2,eno2] = f_counter(period,eni);
%
% Graphical Postprocessing
figure(4311);
t=0:NoS-1;
subplot(211);
plot(t,eni+1,'k',t,y1,'b',t,eno1,'g'); hold on;
stem(t,eni+1,'k');stem(t,y1,'b');stem(t,eno1,'g'); hold off;
title('cycle based counter model, loop external of counter')
xlabel('clock cycle');
ylabel('en_i_n_,_b_l_k,   en_o_u_t_,_g_r_n,   count_b_l_u');
%
subplot(212);
plot(t,eni+1,'k',t,y2,'b',t,eno2,'g'); hold on;
stem(t,eni+1,'k');stem(t,y2,'b');stem(t,eno2,'g'); hold off;
title('cycle based counter model, loop within counter model')
xlabel('clock cycle');
ylabel('en_i_n_,_b_l_k,   en_o_u_t_,_g_r_n,   count_b_l_u');
```

The single statement "**[*y2,eno2*] = *f_counter(period,eni);***" achieves the same result as the 5 lines of code using *f_counter* or *f_countcl* as combinational logic in a loop. This is possible by using the loop over time axis and memory assignment *state* = *nextstate* within the function *f_counter* as shown in listing 4.2.2.

Listing 4.1.3.2: Function *f_counter.m* can operate as nextstate logic and as FSM

```
function [count,eno,nextstate]=f_counter(period,eni,state)
% initialize state if not available
if exist('state')~=1; state=0; end;
%
for n=1:length(eni);
  % NextState Logic
  if eni(n)==0;
    nextstate = state;
  else
    if state < period-1;
      nextstate = state+1;
    else
      nextstate = 0;
    end;
  end;
  %
  %Output-Logic:
  count(n) = state;                  % Moore output
  eno(n) = (state==period-1) & eni(n); % Mealy output
  %
  % apply active clock edge to latch memory
  state = nextstate;
end; % of loop over n
```

1. **Memory model "*state = nextstate*" must be the last statement in the loop**
2. **Do not assign signals *state* and *nextstate* with time index *n*.**

4.2 Pulse-Width Modulator (PWM) as *Matlab* FSM Model

4.2.1 Evaluate FSM Model

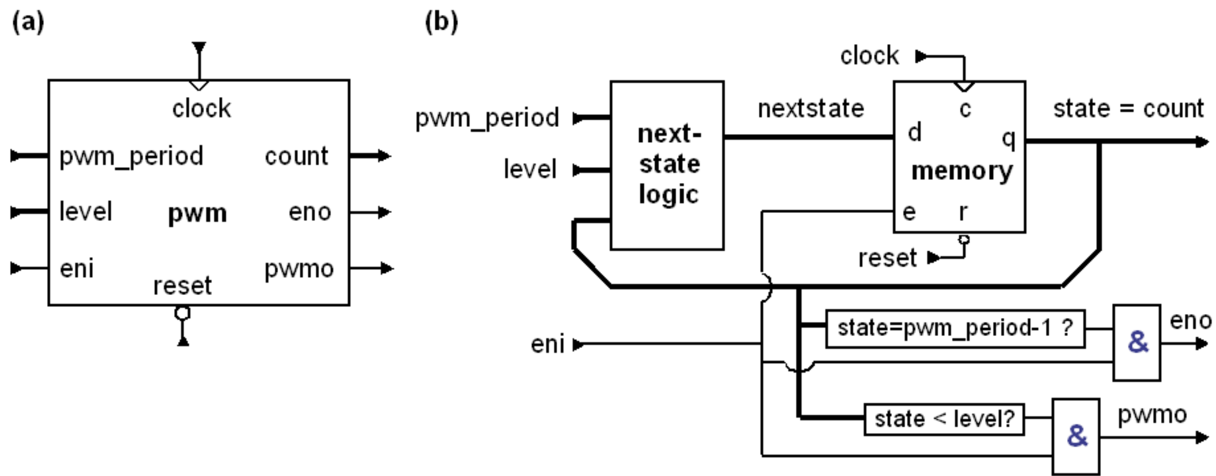


Fig. 4.2.1: pulse-width modulator: **(a)** symbol, **(b)** schematics: *pwmo*=1 needs *state*<*level* .

The pulse-width modulator in Fig. 4.2.1 extends the counter by a *level* input and a PWM output (*pwmo*) signal, whereas *pwmo = 1 when (eni == 1 and state < level).*

Exercise: Assign signal names in Fig. 4.2.1(b) to signals of Fig. 3.3

Stimuli: `pwm_period, level, eni`

Nextstate: `nextstate`

State: `state`

Nextstate Logic: `if eni == 1 then`

`{state = state+1 when state < period-1, else state = 0}`

Moore Outputs: `count`

Mealy Outputs: `eno, pwmo`

Output Logic Moore: `count <= state`

Output Logic Mealy: `eno <= (state == period-1) & eni`

`pwmo <= (state < level) and (eni == 1)`

Which signals have to be initialized like? `state @ t=0,`

`all stimuli (pwm_period, level, eni) for any t`

4.2.2 Testbench for the Pulse-Width Modulator

Listing 4.2.2: Testbench *tb_pwm* testing pulse-width modulator *f_pwm* for two cases: (i) as combinational logic only in a sample-by-sample mode and (ii) over the whole time axis.

```
% Testbench tb_pwm.m
clear all;
% initializations
NoS    = 2001;  % Number of Samples
period = 20;                          % stimuli: max{count}
eni = logical(ones(1,NoS));            % stimuli: input enable
eni(round(NoS/5:NoS/4)) = logical(0); % stimuli: input enable
t=0:NoS-1;
Fg=1/NoS;
level=round((1-cos(2*pi*Fg*t))*period/2);
%level=6*ones(1,NoS);
%
% 1. work off the FSM sample by sample
s=0; % initialize state
for n=1:NoS;
  [pwmf(n),ns] = f_pwm(period,level(n),eni(n),s);
  % apply active clock edge:
  s = ns; % state = nextstate
end; % time loop over time index n
%
% 2. work off simulation module by module
[pwmy] = f_pwm(period,level,eni);
%
% Graphical Postprocessing
figure(432);
subplot(211);
plot(t,period*ones(1,NoS),'m',t,level,'k',t,eni*period/2,'g'); grid on;
xlabel('clock cycle'); ylabel('level,   eni,   period');
subplot(212);
plot(t,pwmf,'b',t,pwmy+1.4,'r',t,eni+0.2,'g'); grid on;
title('pwm output')
xlabel('clock cycle'); ylabel('pwm_f,   eni,   pwm_y');
```

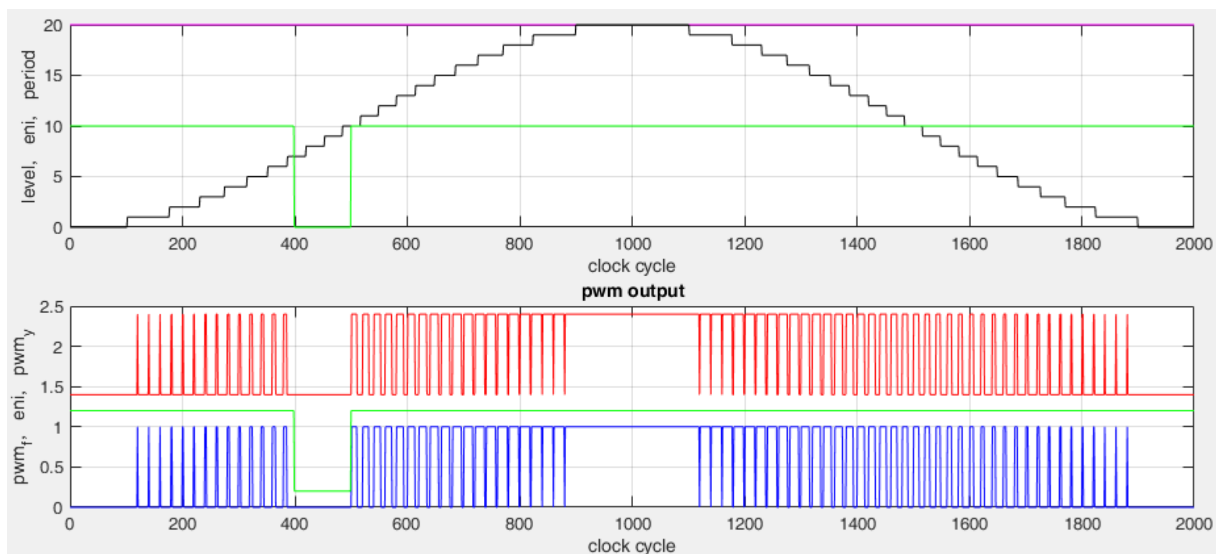


Fig. 4.2.2: Pulse-width modulator results obtained with testbench *tb_pwm*:
Upper subplot: mg: *period*=20, gn: 10**eni*, bk: *level* = (1-cos(…))+offset,
Lower subplot: bl: pmw_x computed sample by sample, gn: *eni*+0.2, rd: pmw_y+1.4.

4.2.3 Exercise: Build a Cycle-Based FSM

Use function *f_pwm* as shown below in file *f_pwm_exercise.m* and modify it such, that it delivers the result shown in Fig. 4.2.2 within testbench *tb_pwm* according to listing 4.2.2.

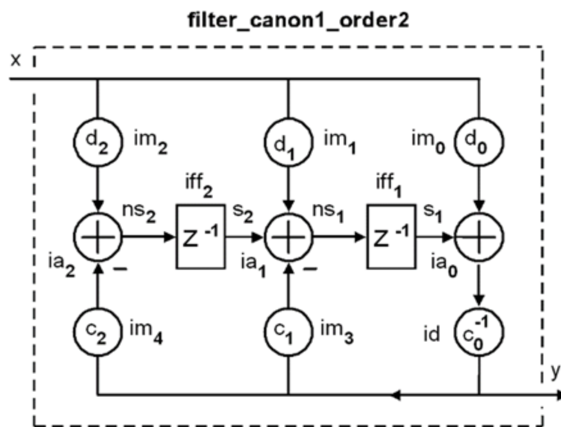
Listing 4.2.3: Function *f_pwm* realizing a pulse-width modulator model. Input vectors *level* and *eni* must have the length of the time axis.

```
function [pwmo,nextstate,count,eno]=f_pwm(period,level,eni,state)
% initialize state if not available
if exist('state')~=1; state=0; end;
%
for n=1:length(eni);
  %
  % NextState Logic
  % ===============
  if eni(n)==0;
% ........................................................
%
    nextstate = state;
% ........................................................
%
  else
% ........................................................
%
    if state < period-1;
% ........................................................
%
      nextstate = state+1;
% ........................................................
%
    else
% ........................................................
%
      nextstate = 0;
% ........................................................
%
    end;
% ........................................................
%
  end;
% ........................................................
%
  %
  %Output-Logic:
  % ============
  % Moore output(s)
  count(n) = state;
% ........................................................
%
  % Mealy output(s)
  eno(n)    = (state==period-1)  & eni(n);  % Mealy output
% ........................................................
%
  pwmo(n)   = (state<level(n)) & eni(n)==1; % Mealy output
% ........................................................
  %
  % apply active clock edge to latch memory
  state = nextstate;
% ........................................................
%
end; % of loop over n
```

4.3 Bi-Quadratic (biquad) Filter

4.3.1 Evaluate FSM Model

(a) **(b)**



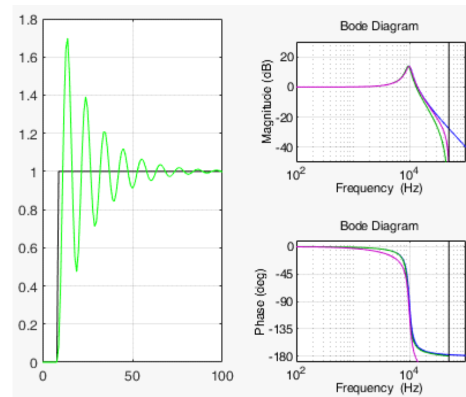


Fig. 4.3.1: (a) Schematic of module *f_filter_canon1_order2*. (b) Results of testbench.

Exercise: Assign signal names in Fig. 4.3.1 to signals of Fig. 3.3.

Stimuli: `x`

Nextstate: `ns1, ns2`

State: `s1, s2`

Nextstate Logic: `ns1 = d1 ·x – c1 ·y + s2`

`ns2 = d2 ·x – c2 ·y`

Moore Outputs: `<none>`

Mealy Outputs: `eno, pwmo`

Output Logic Moore: `<none>`

Output Logic Mealy: `y <= (d0 ·x + s1) / c0`

Which signals have to be initialized like? `state @ t=0,`

`all stimuli (i.e. x) for any t`

Fig. 4.3.1 illustrates a filter to be realized as function *f_filter_canon1_order2*. From

$c_0 \cdot y_n + c_1 \cdot y_{n-1} + c_2 \cdot y_{n-2} = d_0 \cdot x_n + d_1 \cdot x_{n-1} + d_2 \cdot x_{n-2}$

with constants c_k, d_k and k clock periods old samples x_{n-k}, y_{n-k}. Time-domain equation

$y_n = c_0^{-1} \, [d_0 \cdot x_n + d_1 \cdot x_{n-1} + d_2 \cdot x_{n-2} + c_1 \cdot y_{n-1} + c_2 \cdot y_{n-2}]$

corresponds to signal transfer function $STF(z) = \frac{Y(z)}{X(z)} = \frac{d_0 + d_1 z^{-1} + d_2 z^{-2}}{c_0 + c_1 z^{-1} + c_2 z^{-2}}$.

4.3.2 Testbench

Listing 4.3.3: Testbench *tb_filter_canon1_order2* tests function *f_filter_canon1*[*_order2*]

```
% tb_filter.m
clear all;
% Create transfer functions using selfmade LTI data structure
A0=1; D=1/10; f0=1e4; w0=2*pi*f0;
a0=A0*w0^2; a1=0; a2=0; b0=w0^2; b1=2*D*w0; b2=1;
STF_s = [0 2 a0 a1 a2 b0 b1 b2]; % signal transfer function in s-domain
fs    = 1e5; Ts=1/fs; % sampling frequency and sampling interval
STF_z = f_c2d(STF_s,Ts); % CTF(s) -> CTF(z) with Ts=1/fs
%
% Transient Simulation over time axis: tn=n*Ts
NoS = 101;                    % Number ot Samples in time-domain
x=ones(1,NoS); x(1:9) = 0;  % input function (stimulus
%
% 1. work off the FSM sample by sample
s=struct('s',[0 0],'ns',[0 0]); % states of selfmade and of Matlab model
ns=s; % create data structure for nextstate vector
for n=1:NoS;
  [fs(n),ns.s] = f_filter_canon1(STF_z,x(n),s.s); % selfmade
  [fm(n),ns.s] = filter(STF_z(3:5),STF_z(6:8),x(n),s.s); % Matlab
  s = ns; % apply active clock edge: nextstate becomes state
end;
%
% 2. work off simulation module by module: ys selfmade, m: matlab function
ys = f_filter_canon1(STF_z,x); % self-computed y, single statement
ym = filter(STF_z(3:5),STF_z(6:8),x); % Matlab y, single statement
%
% Check results for any sample n:
n=55; fprintf('Check if all 4 values are identical:\n');
fprintf('fs(%d)=%f,  fm(%d)=%f\n',n,fs(n),n,fm(n));
fprintf('ys(%d)=%f,  ym(%d)=%f\n',n,ys(n),n,ym(n));
%
% Graphical Postprocessing
figure(433); subplot(121);
% Time-Domains (=Transient) Simulations:
t=0:NoS-1; plot(t,x,'k',t,ys,'b',t,ym,'r',t,fs,'c',t,fm,'g'); grid on;
% Compute LTI system based on Matlab's and on selfmade s -> z
TFS_s=tf([a2 a1 a0], [b2 b1 b0]); % declare Matlab system TFS(s)
TFS_z_matlab=c2d(TFS_s,Ts);        % translation s -> z by Matlab
TFS_z_selfmade=tf(STF_z(3:5),STF_z(6:8),Ts); % using selfmade s -> z
% Frequency Domain (=Spectral) Graphics
o=bodeoptions; o.FreqUnits='Hz'; o.grid='on';
subplot(222); o.MagVisible='on'; o.PhaseVisible='off'; o.YLim=[-50 30];
bodeplot(TFS_s,'b',TFS_z_selfmade,'g',TFS_z_matlab,'m',o);
subplot(224); o.MagVisible='off'; o.PhaseVisible='on'; o.YLim=[-190 10];
bodeplot(TFS_s,'b',TFS_z_selfmade,'g',TFS_z_matlab,'m',o);
```

Explanations to the Testbench in Listing 4.3.2

- Clearing the *Matlab Workspace*
  Command *clear all* clears the workspace to prevent accidently available variables to interfere.
- Creation of a signal transfer function *STF(s)*
  Next we define the second order system $STF(s) = \frac{A_0\omega_0^2}{\omega_0^2 + 2D\omega_0 s + s^2} = \frac{a_0 + a_1 s + a_2 s^2}{b_0 + b_1 s + b_2 s^2}$.
  We save the self-defined LTI system structure in vector *STF_s* = [T_s *R* $a_0\ a_1\ a_2$ $b_0\ b_1\ b_2$] with T_s=0 making the system time-continuous and order *R*=2.
- Selfmade translation *STF(s)* → *STF(z)* = [T_s *R* $d_0\ d_1\ d_2$ $c_0\ c_1\ c_2$]
  Function *f_c2d* needs sampling interval T_s=1/f_s to translate system *STF_s* to systen *STF(z)* = [T_s *R* $d_0\ d_1\ d_2$ $c_0\ c_1\ c_2$] which defines system *STF(z)_*.
- Translation to *Matlab LTI* systems for plotting
  In section "graphical postprocessing" vectors *STF*_s and *STF*_z are translated to the *Matlab LTI* systems *TFS*_s and *TFS*_z, respectively, to be used in command *bodeplot*.
- Declaration of time axis
  Parameter *NoS* declares the number of samples to be used for transient simulation on the time axis $t_n = n \cdot T_s$. This time axis is declared in section "graphical postprocessing" as vector *t* as abscissa in *Matlab's plot* command.
- Work off the FSM sample by sample by looping index (*n*) over time axis :
  ys and ***ym*** are vectors containing the transient response to stimulus *x* computed with our selfmade function *f_filter_canon1_order2* and Matlab's function *filter*, respectively, computed sample by sample. The results for *fs* and *fm* should be identical.
- Transient simulation with single statement for the entire time axis:
  fs and ***fm*** are vectors containing the transient response to stimulus *x* computed with our selfmade function *f_filter_canon1_order2* and Matlab's function *filter*, respectively, with a single statement over the entire time axis. The results for *fs* and *fm* should be identical. (Check for Matlab command *filter* with *help filter*.)
- Check to validate your model:
  Choose any sample *n*, in the example code it is *n*=55, and assert that all 4 output values *ys, ym, fs, fm* are identical: (i) you should see 4 identical real number printed on your screen, and (ii) you should see in the left hand side of the graphics the stimulus signal *x* as black step and the green line of *fm* covering all the other lines made with *ys, ym* and *fs*.
- Graphical Postprocessing:
  Transient diagram:
  Graphical postprocessing should look like Fig. 4.3.1(b). Explanations are given above: All 4 output curves in the transient left hand side of the figure, i.e. *fs*, *fm*, *ys* and *ym*, should be identical. Therefore, only the green step response is seen because it covers all other curves.
  Bode diagram:
  Blue curve shows the Bode diagram of the time continuous transfer function, Green curve shows time-discrete TF computed with selfmade f_c2d from time-continuous Magenta curve shows Bode diagram of the time discrete TF computed with Matlab's c2d. For experts: Matlab command bodeplot was instead of bode, allowing for options like frequency axis scaling in Hz.

4.3.3 Exercise: Build and Test a 2nd Order IIR Filter

Run testbench *tb_filter.m*. Should work!
In file *tb_filter.m* change all function calls *f_filter_canon1* to *f_filter_canon1_order2.m*.
Copy file *f_filter_canon1_order2_exercise.m* to file *f_filter_canon1_order2.m*.
Complete file *f_filter_canon1_order2.m* such, that *tb_filter.m* delivers the original results.

Listing 4.3.3: Function *f_filter_canon1_order2* modeling a 2nd order IIR filter in first canonical direct structure.

```
function [y,ns] = f_filter_canon1_order2(STF_z,x,s)
if exist('s')~=1; s=zeros(1,2); end; % default for missing state
Ts=STF_z(1); % sampling interval
assert(Ts~=0,'TF must  NOT have Ts=0 as this is time-continuous');
R=STF_z(2);    % filter order
d0=STF_z(3);   % numerator coefficient d0
d=STF_z(4:5); % numerator coefficients
c0=STF_z(6);   % denominator coefficient c0
c=STF_z(7:8); % denominator coefficients
%
% loop over time axis
for n=1:length(x);
  % evaluate y(n) fist, because it is needed for nextstate computation
  y(n) = (d0*x(n) + s(1))/c0;
  %
  % compute next state
  ns(2) = x(n)*d(2) - y(n)*c(2);
  ns(1) = x(n)*d(1) - y(n)*c(1) + s(2);
  %
  % apply active clock edge for the case n < length(x)
  s = ns;
end; % of loop over n
```

4.3.4 Exercise: Test Your 2nd Order Filter within *tb_DCDCbuck_pcb00.m*

Run testbench *tb_DCDCbuck_pcb00.m*. Should work!
In this file change function calls *f_filter_canon1* → *f_filter_canon1_order2_exercise*.
Exception: Change function *f_filter_canon1*(*CTF_z,...*) only, if vector length of *CTF_z* is 8.
(*PTF_z* and *QTF_z* will always have 8 elements.)
Run testbench *tb_DCDCbuck_pcb00.m* again, now with your own filter model. Should work!

4.4 Successive FSMs in Feed-Forward Situation

4.4.1 Evaluate FSM Model

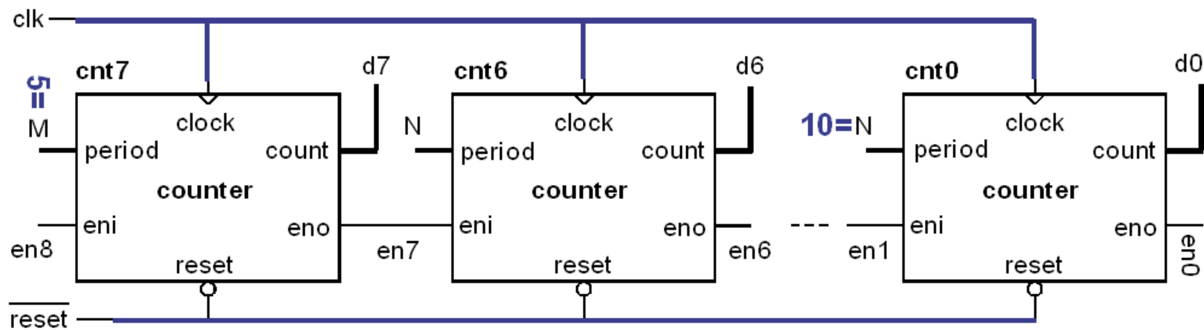


Fig. 4.4.1: Schematic of 7-stage counter made of counter elements of Fig. 4.1.1.

Exercise: Assign signal names in Fig. 4.4.1 to signals of Fig. 3.3.

Stimuli: M, N, en8

clock and reset are applied to the memory and are no stimuli

Nextstate: <not visible in Fig. 4.3.1>

State: d0, d1, … , d7 (known from Fig. 4.1.1)

Nextstate Logic: counter logic with en(#) = en(#+1), #=0…7

Moore Outputs: d(#), #=0…7 (known from Fig. 4.1.1)

Mealy Outputs: en(#), #=0…7 (known from Fig. 4.1.1)

Which signals have to be initialized like? state @ t=0,

all stimuli (M, N, en8) for any t

4.4.2 Testbench

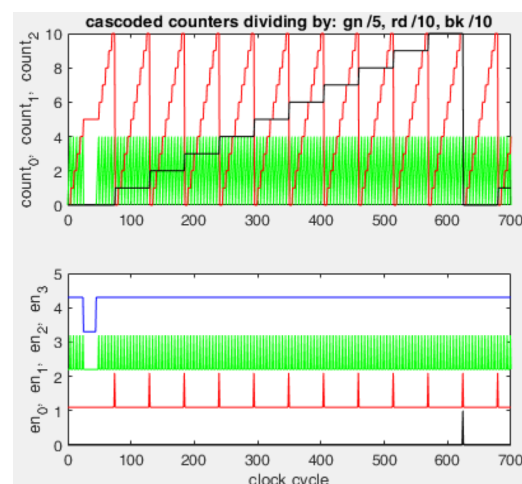
Listing 4.4.2: Testbench *tb_counterchain.m*: dividing clock signal by $M \cdot N^2$ using *f_counter*

```
% Testbench tb_counterchain.m
% initializations
clear all;
NoS     = 701;  % Number of Samples
periodM =  5;                % stimuli: max{count}
periodN = 10;                % stimuli: max{count}
eni = logical(ones(1,NoS)); % stimuli: input enable
eni(26:45) = logical(0);     % stimuli: input enable
%
s=struct(... % s is the total state, "...": continue statement next line
          's0', 0,... % declare and initialize state0 of total state
          's1', 0,... % declare and initialize state1 of total state
          's2', 0);   % declare and initialize state2 of total state
 ns=s; % nextstate vector takes data structure from state
% 1. work off the FSM sample by sample
 for n=1:NoS;
   % nextstate logic
   [f2(n),enf2(n),ns.s2] = f_counter(periodM,eni(n),s.s2);
   [f1(n),enf1(n),ns.s1] = f_counter(periodN,enf2(n),s.s1);
   [f0(n),enf0(n),ns.s0] = f_counter(periodN,enf1(n),s.s0);
   %
   % state moemory: apply active clock edge
   s = ns; % state = nextstate
 end; % time loop over time index n
%
% 2. work off simulation module by module
[y2,eny2] = f_counter(periodM,eni);
[y1,eny1] = f_counter(periodN,eny2);
[y0,eny0] = f_counter(periodN,eny1);
%
% Graphical Postprocessing
figure(43);
t=0:NoS-1;
subplot(221); plot(t,f2,'g',t,f1,'r',t,f0,'k');
ylabel('count_0,  count_1,  count_2');
title('cascoded counters, FSM loops within counters');
subplot(223); plot(t,eni+3.3,'b',t,enf2+2.2,'g',t,enf1+1.1,'r',t,enf0,'k');
xlabel('clock cycle');
ylabel('en_f_0,  en_f_1,  en_f_2,  en_i_n');
subplot(222); plot(t,y2,'g',t,y1,'r',t,y0,'k');
ylabel('count_y_0,  count_y_1,  count_y_2');
title('cascoded counters, FSM loops around counters');
subplot(224); plot(t,eni+3.3,'b',t,eny2+2.2,'g',t,eny1+1.1,'r',t,eny0,'k');
xlabel('clock cycle');
ylabel('en_y_0,  en_y_1,  en_y_2,  en_i_n');
```

Fig. 4.4.2:
Three cascaded counters according to listing 4.4.2 dividing the clock by green: 5, red: 10, black: 10. The

Top subplot: illustrates count states, the

Bottom subplot illustrates enable-flags with en3 (blue) being the front-end input enable.



The 1st block of bold code lines in testbench *tb_counterchain*, listing 4.4.2, realizes an FSM with all memories being clocked at the same time point. This is performed by the statement "***s* = *ns***", whereas state ***s*** and nextstate ***ns*** are realized as structs with fields *s0*, *s1*, *s2* corresponding to the state and nextstate of counters 0, 1, 2, respectively.

Interesting *Matlab* features: composition of a struct. Here we need to continue statements in the next line indicated by three dots "…" in Matlab.

The 2nd block of bold code lines in testbench *tb_counterchain*, listing 4.4.2, works off the time axis counter by counter. This does not correspond to synchronous clocking of all memory but delivers correct results because this counter chain is a pure feed-forward structure.

4.4.3 Exercise *Counterchain*: Successive FSMs

Open file *tb_counterchain_exercise.m* delivered by the author and complete statements

```
% Testbench tb_counterchain.m
% initializations
clear all;
NoS     = 701;  % Number of Samples
periodM =  5;                 % stimuli: max{count}
periodN = 10;                 % stimuli: max{count}
eni = logical(ones(1,NoS)); % stimuli: input enable
eni(26:45) = logical(0);     % stimuli: input enable
%
 s=struct(... % s is the total state, "...": continue statement next line
          's0', 0,... % declare and initialize state0 of total state
          's1', 0,... % declare and initialize state1 of total state
          's2', 0);   % declare and initialize state2 of total state
 ns=s; % nextstate vector takes data structure from state
 % 1. work off the FSM sample by sample
 for n=1:NoS;
   % nextstate logic
   [f2(n),enf2(n)       ] =
   [f1(n),              ] =
   [                    ] =
   %
   % state moemory: apply active clock edge
   s = ns; % state = nextstate
 end; % time loop over time index n
%
% 2. work off simulation module by module
[y2,eny2] = f_counter(periodM,     );
[y1,eny1] = f_counter(             );
[y0,eny0] =
%
% Graphical Postprocessing
figure(43);
t=0:NoS-1;
subplot(221); plot(t,f2,'g',t,f1,'r',t,f0,'k');
ylabel('count_0,  count_1,  count_2');
title('cascoded counters, FSM loops within counters');
subplot(223); plot(t,eni+3.3,'b',t,enf2+2.2,'g',t,enf1+1.1,'r',t,enf0,'k');
xlabel('clock cycle');
ylabel('en_f_0,  en_f_1,  en_f_2,  en_i_n');
subplot(222); plot(t,y2,'g',t,y1,'r',t,y0,'k');
ylabel('count_y_0,  count_y_1,  count_y_2');
title('cascoded counters, FSM loops around counters');
subplot(224); plot(t,eni+3.3,'b',t,eny2+2.2,'g',t,eny1+1.1,'r',t,eny0,'k');
xlabel('clock cycle');
ylabel('en_y_0,  en_y_1,  en_y_2,  en_i_n');
```

4.5 Successive FSMs in Non-Linear Feedback Situations

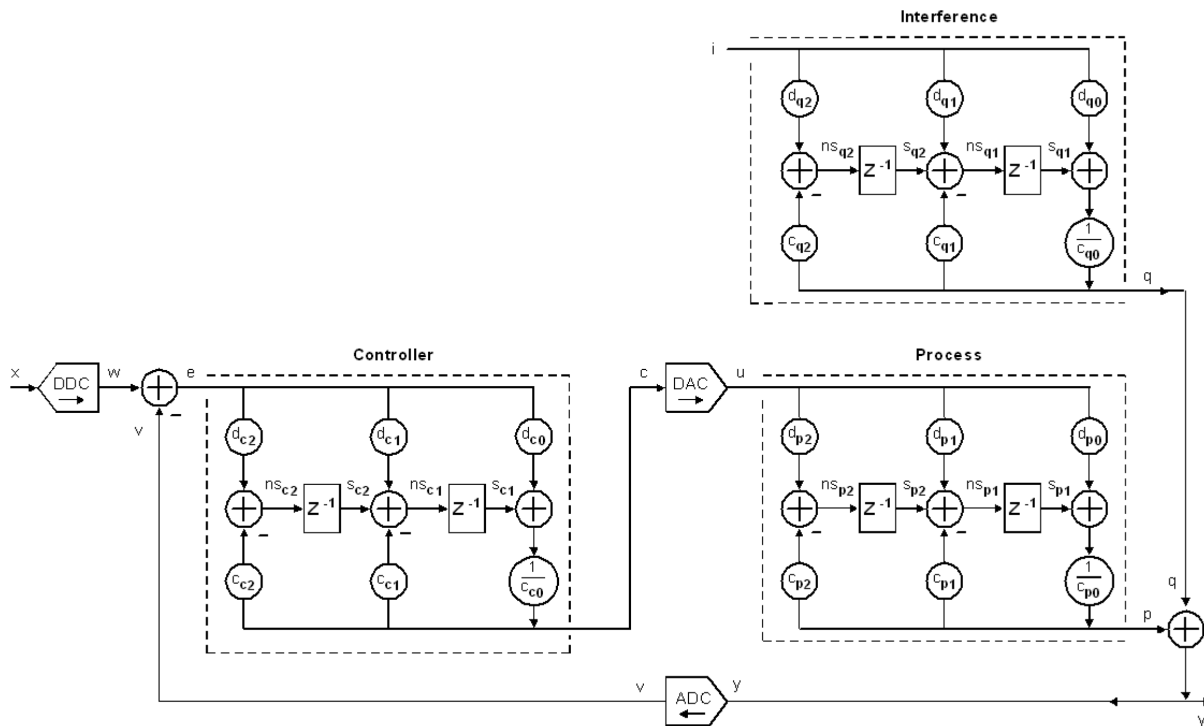
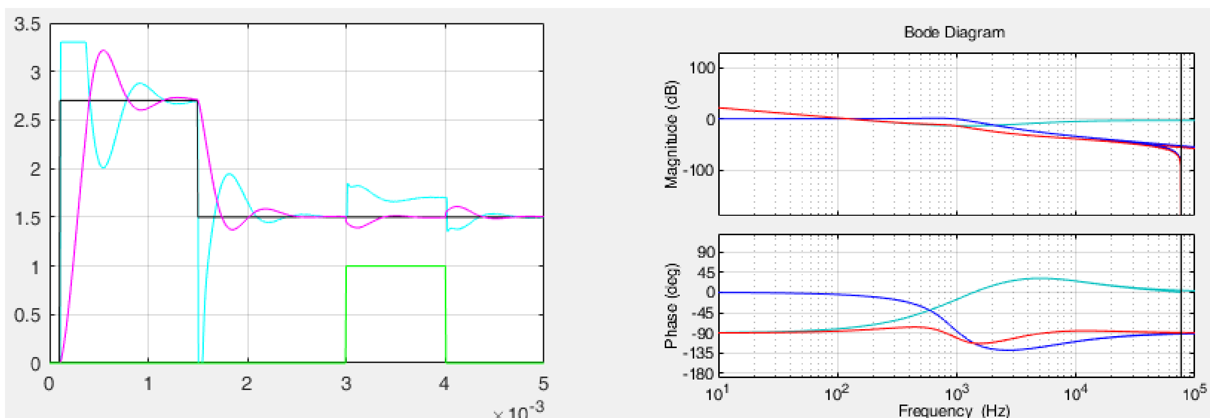


Fig. 4.5.1: Time-discrete DC/DC buck converter model

Fig. 4.5.1 illustrates the feedback loop of the DC/DC buck converter model. The PID controller is modeled as biquad filter *CTF*(z). To simulate the setup on the discrete time axis, also the *RLC* lowpass is modeled in z, consisting of process transfer function *PTF*(z) and inference (quarrel) transfer function *QTF*(z). Nonlinear saturation of A/D and D/A converter output is respected. Simulation of this setup as FSM requires working off the time axis sample by sample in a loop outside of functions that model *CTF*, *PTF and QTF*.



(a) black: x=w, cyan: u, green: i, magenta: y , **(b)** blue: *PTF*, cyan *CTF*, red: (*CTF*PTF*)

Fig. 4.5.2: Time-discrete DC/DC buck converter simulation obtained with Listing 4.5.

Listing 4.5: Matlab FSM model of DC/DC buck controller illustrated in Fig. 4.5.1.

```
% tb_DCDCbuck_bcb00.m
clear all;
% Example model parameters
% Frequencies
fclock_50M=50e6; Tclock_50M=1/fclock_50M; % system clock 50 MHz
pwm_period=330;Ts=Tclock_50M*pwm_period; fs=1/Ts; % sampling fcequency
fprintf('\ntb_DCDC_buck Converter\n',fclock_50M/1e6);
fprintf('System clock: %i MHz, fs=%d KHz \n',fclock_50M/1e6,fs/1e3);
% RLC Lowpass
Rg=150e-3; Rw=50e-3; RD=(Rg+Rw); L=43e-6; RL=1e6; GL=1/RL;
RC=50e-3; C=680e-6;
Ap0=1/(1+RD*GL); Dp=0.5*(RC+RD)*sqrt(C/L)+0.5*GL*sqrt(L/C); fp0=1/sqrt(4*pi^2*L*C);
fpn1=1/(2*pi*RC*C); fqn2 = RD/L;
fpp1=fp0*abs(Dp-sqrt(Dp*Dp-1)); fpp2=fp0*abs(Dp+sqrt(Dp*Dp-1));
fprintf('Lowpass: A0=%d, D=%d, f0=%d KHz\n',Ap0,Dp,fp0/1e3);
fprintf('fpp1=%d Hz, fpn1=%d Hz, fqn2=%d KHz, fpp2=%d KHz \n',...
         fpp1,       fpn1,       fqn2/1e3,    fpp2/1e3);
%Controller
Kcp=25; fcp1=0; fcn1=fpp1;fcn2=fs/4; fcp2=fcn2*1e2; % Controller
fprintf('Controller: Kcp=%d\n',Kcp);
fprintf('fcp1=%d Hz, fcn1=%d  Hz, fcn2=%d KHz, fcp2=%d KHz \n',...
         fcp1,       fcn1,       fcn2/1e3,    fcp2/1e3);
%
% Create transfer functions using tf and zpk
R=2; % oRder of controller tranfer function
wn1=2*pi*fcn1; wn2=2*pi*fcn2; wp1=2*pi*fcp1; wp2=2*pi*fcp2;
%CTF_s=[0,2, Kcp*[wn1*wn2/(wn1+wn2),1,1/(wn1+wn2), wp1*wp2/(wp1+wp2),1,1/(wp1+wp2)]];
KP=0.2; KI=763.8; KD=1.02e-5; wp2=50e3;
CTF_s=[0,2,   KI*wp2, KI+KP*wp2 ,KP+KD*wp2,    1e-12, wp2, 1];
PTF_s=[0,2, 1,C*RC,0, 1+RD*GL,(RC+RD+RD*RC*GL)*C+GL*L,(1+RC*GL)*L*C];%Process
QTF_s=[0,2, -RD,-(RC*RD*C+L),-RC*L*C, PTF_s(6:8)]; % Inference TF
CTF_z = f_c2d(CTF_s,Ts); % s->z with sampling fcequency fs
PTF_z = f_c2d(PTF_s,Ts); % s->z with sampling frequency fs
QTF_z = f_c2d(QTF_s,Ts); % s->z with sampling frequency fs
%
% DAC and ADC
Udamin=0; Udamax=3.3; Ndamin=0; Ndamax=pwm_period;
delta0=0, delta1=(Udamax-Udamin)/(Ndamax-Ndamin);
delta=[delta0 delta1]; bda=[Udamin Udamax]; % specify dac
Uadmin=0; Uadmax=4.095; Nadmin=0; Nadmax=4095;
alpha0=0; alpha1=(Nadmax-Nadmin)/(Uadmax-Uadmin);
alpha=[alpha0 alpha1]; bad=[Nadmin Nadmax]; % specify adc
fprintf('DAC: PWM_period=%i bits, Uoutmin=%d, Uoutmax=%d \n',...
              pwm_period,          Udamin,      Udamax);
%
% Transient Simulation over time axis: tn=n*Ts
Tend=5e-3; Txstepup=0.1e-3; Txstepdn=1.5e-3; Tistepup=3e-3; Tistepdn=4e-3;
NoS = ceil(Tend/Ts); % Number ot Samples in time-domain
fprintf('Tend = %d ms, NoS = %i samples\n',Tend*1e3,NoS);
nxstepup = ceil(Txstepup/Ts); nxstepdn=ceil(Txstepdn/Ts);
x=2.7*ones(1,NoS); x(1:nxstepup)=0; x(nxstepdn:end)=1.5; % input function (stimulus)
nistepup=ceil(Tistepup/Ts); nistepdn=ceil(Tistepdn/Ts);
i=zeros(1,NoS);i(nistepup:nistepdn)=1;  % load current
%
% 1. work off the FSM sample by sample
s=struct('c', [0 0], ... % state of controller
         'p', [0 0], ... % state of process
         'q', [0 0], ... % state of quarrel (inference)
         'adc', 0,   ... % state of ADC
         'dac', 0  );    % state of DAC
ns=s; % create data structure for nextstate vector
for n=1:NoS;
  w = f_adc(x(n),alpha,bad);  % wanted input: after A/D-converted input voltage x
  v = s.adc; % process Variable:  A/D-converted output voltage y
  e = (w-v); % difference point's output
  [c(n),ns.c] = f_filter_canon1(CTF_z,e,s.c); % controller
  u(n) = s.dac;  ns.dac = f_dac(c(n),delta,bda); % DAC
  [p,ns.p] = f_filter_canon1(PTF_z,u(n),s.p); % process
  [q,ns.q] = f_filter_canon1(QTF_z,i(n),s.q); % quarrel (inference)
  y(n) = p+q;
  ns.adc = f_adc(y(n),alpha,bad);  % feeding output voltage into feedback ADC
  s = ns; % latch nextstate vector into memeory
end;
%
% Graphical Postprocessing
t = [0:NoS-1]*Ts;
figure(45);
subplot(121); % Time-Domain Model
plot(t,u,'c',t,x,'k',t,y,'m',t,i,'g'); grid on; xlim([0 Tend]);%ylim([-0.7 3.3]);
subplot(122); % Frequency Domain Model
TFC_s=tf(f_h(CTF_s(3:R+3)),f_h(CTF_s(R+4:2*R+4)));
TFC_z=tf(    CTF_z(3:R+3),      CTF_z(R+4:2*R+4),Ts);
TFP_s=tf(f_h(PTF_s(3:5)),f_h(PTF_s(6:8)));TFP_z=tf(PTF_z(3:5),PTF_z(6:8),Ts);
TFQ_s=tf(f_h(QTF_s(3:5)),f_h(QTF_s(6:8)));TFQ_z=tf(QTF_z(3:5),QTF_z(6:8),Ts);
o=bodeoptions; o.FreqUnits='Hz'; o.grid='on'; o.Ylim=[-190 130];
bodeplot(TFC_s,'c',TFP_s,'b',TFC_s*TFP_s,'r',o);
hold on; bodeplot(TFC_z,'c',TFP_z,'b',TFC_z*TFP_z,'r',o); hold off;
```

4.5.1 Exercise: Match your Models to Your *DCDCbuck* Board

Check for the PCB number (##) of your DCDCbuck board.
Copy file *tb_DCDCbuck_pcb00.m* to *tb_DCDCbuck_pcb##.m*.
Change device parameters to those of your PCB and simulate it.
Compare Simulations with *Simulink*, *LTspice* and *Matlab* to your board ##.
Try to get matching between the behavior of your board and your models of it
.

5 Conclusions

5.1 Summary

This tutorial demonstrates how to

- Use *Matlab* to model and simulate LTI systems in both *s* and *z* plane.
- Translate LTI systems in *s* and *z* to finite difference equations,
- Translate finite difference equations to a finite state machine (FSM) model,
- Simulate a cycle-based FSM with textual *Matlab* commands.

5.2 Key Skills Learned

Key skills concerning LTI systems:
Declare, modify and plot an LTI system model with *Matlab*:

- Create *Matlab* LTI systems as transfer functions in *s* and *z*.
- Plot *Bode* diagram, step and impulse responses of *Matlab* LTI systems.
- Translate an LTI system in *s* into an LTI system in *z*.
- Translate an LTI system in *z* to finite difference equations,
- Translate finite difference equations into cycle-based FSM model.

6 References

[1] *Matlab*, available: http://www.mathworks.de/products/slhdlcoder/.
[2] *Open SystemC Initiative (OSCI)*, available: http://www.systemc.org/news/events/
[3] *VHDL*, available https://en.wikipedia.org/wiki/VHDL
[4] *Verilog*, available https://en.wikipedia.org/wiki/Verilog .
[5] *Octave*, available http://octave.sourceforge.net/
[6] Scilab, available http://de.wikipedia.org/wiki/Scilab
[7] David Houcque, Introduction to Matlab for Engineering Students, Nothwestern University, 2005, available : https://www.mccormick.northwestern.edu/documents/students/undergraduate/introduction-to-matlab.pdf.
[8] M. Schubert, Linear Feedback Loops, OTH Regensburg 2018, available : https://hps.hs-regensburg.de/scm39115/homepage/education/lessons/LinearFeedbackLoops/LinearFeedbackLoops.pdf.

7 Appendix: PID Controller Realized as Biquad Filter

7.1 PID Design

7.1.1 0th Order P Controller

A proportional (P) controller has the transfer function (TF)

$$CTF(s) = K_p . \qquad (7.1.1)$$

7.1.2 1st Order PI Controller

An integral part (I) in the controller to obtains high static gain delivers the PI controller TF

$$CTF(s) = K_p + \frac{K_I}{s+\omega_{p1}} = \frac{(K_p\omega_{p1}+K_I)+K_p s}{\omega_{p1}+s} = \frac{a_0+a_1 s}{b_0+b_1 s} \qquad (7.1.2)$$

with $a_0 = K_p\omega_{p1} + K_I$, $a_1 = K_p$, $b_0 = \omega_{p1}$, $b_1 = 1$. For an ideal integrator the pole was $\omega_{p1} = 0$, but some simulators (e.g. *LTspice*) do not allow for infinite DC amplification, so that any tiny $\omega_{p1} > 0$ is required for operating point computation when using *Laplace* models.

7.1.3 2nd Order PID Controller

Frequently we have to add a differential part (D) for stability reasons, obtaining the PID controller transfer function

The total controller transfer function evaluates with ω_{p1}=0 to

$$CTF(s) = K_P + \frac{K_I}{s} + K_D \cdot \frac{s}{1+s/\omega_{p2}} = \frac{K_I\omega_{p2} + (K_I + K_P\omega_{p2})\cdot s + (K_P + K_D\omega_{p2})\cdot s^2}{0 + \omega_{p2}\cdot s + s^2} \qquad (7.1.3)$$

Ansatz $CTF(s) = \dfrac{a_{c0} + a_{c1}\cdot s + a_{c2}\cdot s^2}{b_{c0} + b_{c1}\cdot s + b_{c2}\cdot s^2}$ delivers (7.1.4)

(7.1.5)

$$a_{c0} = K_I \cdot \omega_{p2} \qquad a_{c1} = K_I + K_P\cdot\omega_{p2} \qquad a_{c2} = K_P + K_D\cdot\omega_{p2}$$

$$b_{c0} = 0 \qquad b_{c1} = \omega_{p2} \qquad b_{p2} = 1$$

A tiny b_0 may be used when $\omega_{p1} > 0$ is required, e.g. for a *Laplace* model with *LTspice*.

7.2 Pole-Zero Design

Other people prefer to define minimum amplification K_P plus zeros ω_{n1} , ω_{n2} with $\omega_{n1} \le \omega_{n2}$ and poles ω_{p1} , ω_{p2} with $\omega_{p1} \le \omega_{p2}$ and typically ω_{p1}=0. The computation of the time-continuous transfer function according to:

$$CTF(s) = const \cdot \frac{(s+\omega_{n1})(s+\omega_{n2})}{(s+\omega_{p1})(s+\omega_{p2})} = K_p \frac{\frac{\omega_{n1}\omega_{n2}}{\omega_{n1}+\omega_{n2}} + s + \frac{1}{\omega_{n1}+\omega_{n2}} s^2}{\frac{\omega_{p1}\omega_{p2}}{\omega_{p1}+\omega_{p2}} + s + \frac{1}{\omega_{p1}+\omega_{p2}} s^2} \tag{7.2.1}$$

Comparison with (7.1.4) delivers

$$a_{c0} = K_p \frac{\omega_{n1}\omega_{n2}}{\omega_{n1}+\omega_{n2}}, \qquad a_{c1} = K_p, \qquad a_{c2} = \frac{K_p}{\omega_{n1}+\omega_{n2}},$$

$$b_{c0} = \frac{\omega_{p1}\omega_{p2}}{\omega_{p1}+\omega_{p2}}, \qquad b_{c1} = 1, \qquad b_{c2} = \frac{1}{\omega_{p1}+\omega_{p2}}. \tag{7.2.2}$$

7.3 Comparison

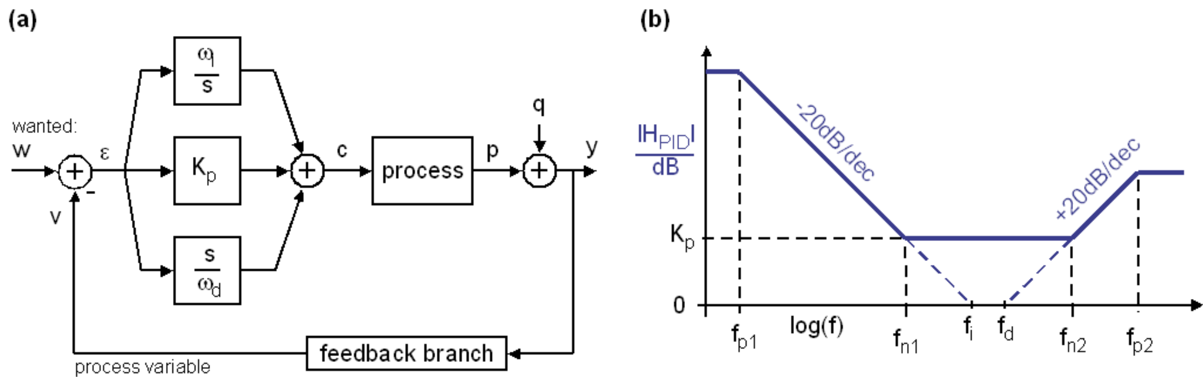


Fig. 7.3.1: PID controller within a loop, (b) Relations between frequencies and K_I, K_P, K_D.

Fig 7.3.1 illustrates a PID controller schematics with $\omega_I = K_I$ and $\omega_D = 1/K_D$.

For the typical setting $\omega_{p1} = 0$ both methods explained above deliver the same denominator. The accurate relation between zeros and constants K_X is

$$\omega_{n1,2} = \frac{-K_P}{2K_D}\left(1 \pm \sqrt{1 - \frac{4K_I K_D}{K_P^2}}\right) \tag{7.3.1}$$

Approximately we can say that $\omega_{n1} \sim \omega_I/K_P = K_I/K_P$ and $\omega_{n2} \sim \omega_D K_P = K_P/K_D$.

Fig. 7.3.2 illustrates Ma phase-margin $\phi_M = 45°$ setting for an oscillating lowpass, i.e. D << ½.

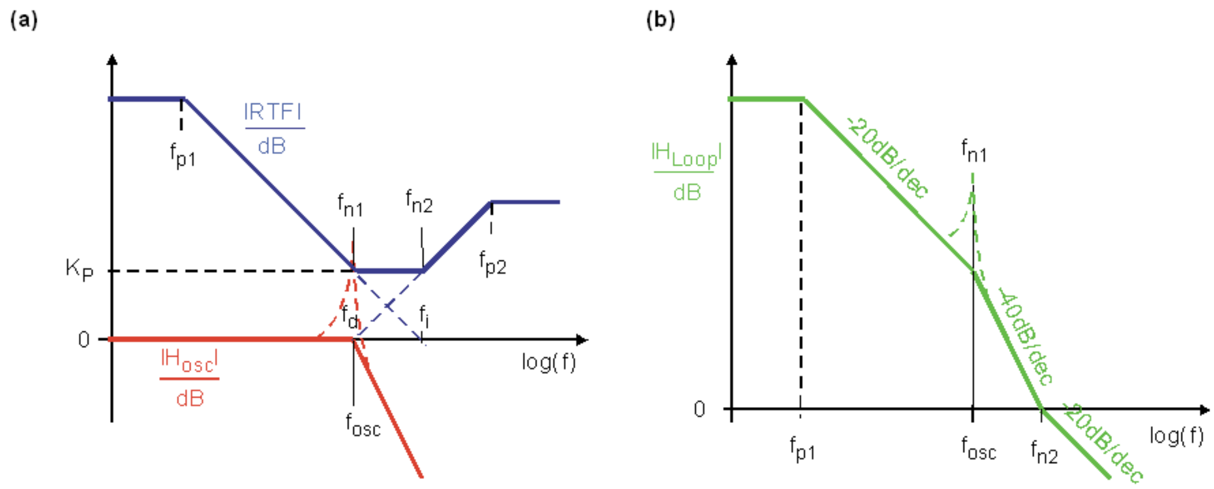


Fig. 7.3.2: PID controller within a loop, (b) Relations between frequencies and K_I, K_P, K_D.