Daftar Pustaka

- Ross, T. J. 2004. *Fuzzy Logic with Engineering Application, Second Edition.* Wiley John & Sons, Ltd.
- Ana Ratnawati ST, M.Eng, Dwi. Modul Kuliah Sistem Fuzzy.
- *www.wileyeurope.com/go/fuzzylogic*
- Astuti, Ir. Budi. 1999. *Teknik Tenaga Listrik.* Yogyakarta: Jurusan Teknik dan Manajemen Industri Fakultas Teknologi Industri Universitas Islam Indonesia.
- Ogata, Katsuhiko. 1997. *Teknik Kontrol Automatik*, Jilid 1, Edisi Kedua. Jakarta : Erlangga.
- Modul Pelatihan MATLAB. 2009. Yogyakarta : Laboratorium Kendali dan Mikroprosesor Jurusan Teknik Elektro Fakultas Teknologi Industri Universitas Islam Indonesia.
- C. Kuo, Benjamin, 1995. *Teknik Kontrol Automatik*, Edisi Bahasa Indonesia, Jilid 1. Jakarta : PT. Prenhallindo.
- Kusamadewi, Sri, 2002. *Analisis* dan *Desain Sistem Fuzzy Menggunakan Tool Box Matlab.* Yogyakarta : Graha Ilmu.
- Kusumadewi, Sri, & Purnomo, Hari. 2004. *Aplikasi Logika Fuzzy untuk Pendukung Keputusan.* Yogyakarta : Graha Ilmu.
- Paulus S.Si., M.kom, Erick, & Nataliani, S.Si., Mkom, Yessica. 2007. *GUI Matlab.* Yogyakarta : Penerbit Andi.

- Rifky Indriarto, Muhammad 2006. "SIMULASI KENDALI KECEPATAN MOTOR DC BERBASIS ALGORITMA ANFIS". Skripsi, tidak diterbitkan. Yogyakarta : Fakultas Teknologi Industri Jurusan Teknik Elektro Universitas Islam Indonesia .
- Wiryadinata, Romy 2005." SIMULASI JARINGAN SYARAF TIRUAN BERBASIS METODE *BACK PROPAGATION* SEBAGAI PENGENDALI KECEPATAN MOTOR DC". Skripsi, tidak diterbitkan. Yogyakarta : Fakultas Teknologi Industri Jurusan Teknik Elektro Universitas Islam Indonesia .
- Alim Kurniafian, Fahmi 2008. "SIMULASI KENDALI SISTEM TANGKI AIR

  MENGGUNAKAN PENGENDALI PID DAN *FUZZY*". Skripsi, tidak diterbitkan. Yogyakarta : Fakultas Teknologi Industri Jurusan Teknik Elektro Universitas Islam Indonesia .

LAMPIRAN

```
[illegible]
[illegible] PERCOBAAN DENGAN 2INPUT 1OUTPUT 3MF R=3 aturan 189 [illegible]

The program plots the membership functions for the two inputs
as well as calculating the percent difference between the
redicted output and the actual output.
[illegible]

Initialize variables

ear
:
The data set Z

01;                    % Number of data tuples used in this code

d inpelatihan5.mat;            % Number of data tuples can be increased


errorpelatihan5,derrorpelatihan5]';     % Save this for testing the fuzzy system later
ve=X;

n X the columns are the input vectors and the outputs are

d outpelatihan5.mat;            % Number of data tuples can be increased

utpelatihan5;
ve=Y;    % Save tis for testing the fuzzy system later

ad=200;

px=X;
py=Y;

 k=2:Ngrad
 X=[X tempx];     % Makes columns that are the input portions
 Y=[Y; tempy];  % Makes one long column


et number of inputs and rules

;
;

ext set the initial conditions for the Gradient method

e start off by selecting the output membership function centers Bpast
nd initiate the system by using the output membership function centers
or the first 14 data tuples.
-150;
0;
150;


st=[B3 B1 B1];
```

```
Next, pick the *initial* input membership function centers to
lie in the middle of the training data at some type of
regular spacing - this is just a guess at where they should
oe.
1=-1750;
2=0;
3=1741;
1=-650;
2=0;
3=650;


ast=[C11 C13 C13;
      C21 C22 C23];


'inally, pick an initial spread for all the input membership functions.
lere, we simply pick all the intial spreads to be the same.

'mapast=[750 750 750;
  300 300 300];

ote that these "past" values correspond to time zero, but this will
e indexed with time one since Matlab can't use the zero index.

ext, we choose the step sizes for the algorithm.

bda_1=1;  %Size step for the output membership function
bda_2=1;  % Step size for the c parameters in C
bda_3=1;  % Step size for the sigma parameters in Sigma

ext, we have the main loop where we train
he fuzzy system.

 k=1:Ngrad*M  % For all the training data, now we have M*Ngrad data points
um(k)=0;
ompute the xi (chi) vectors for the given data pair.

or i=1:R
 mu(i,k)=1;  % Initialization
  for j=1:n
        mu(i,k)=mu(i,k)*exp(-.5*((X(j,k)-Cpast(j,i))/Sigmapast(j,i))^2);
  end
end

umorator(k)=0;
enominator(k)=0;
or i=1:R
 b=Bpast(i);
 numorator(k)=numorator(k)+mu(i,k)*b;
 denominator(k)=denominator(k)+mu(i,k);
nd
(k)=numorator(k)/denominator(k); % Compute the output vector
                                 % for kth training data pair
```

```
den(k)=sum(mu(:,k));
xi(:,k)=mu(:,k)/den(k);

epsilon(k)=f(k)-Y(k);        % For use in the gradient updates

e(k)=0.5*(epsilon(k))^2;      % Compute the approximation error
                             % While often you would, we do not use
                             % this to help specify a termination
                             % condition. Here, we simply train
                             % for a fixed number of steps.

Next, we specify the Gradient equations

First, set a value for which no input membership function spread can
decrease below - call this value sigmabar

mabar=.01;

Next we update the output centers B

 i=1:R

 B(i)=Bpast(i)-lambda_1*epsilon(k)*xi(i,k);
                                % Update consequent parameters

 for j=1:n

     C(j,i)=Cpast(j,i)-lambda_2*epsilon(k)*((Bpast(i)-f(k))/den(k))*...
               mu(i,k)*((X(j,k)-Cpast(j,i))/(Sigmapast(j,i))^2);
                                % Update input mf centers


     Sigma(j,i)=Sigmapast(j,i)-lambda_3*epsilon(k)*((Bpast(i)-f(k))/den(k))*...
               mu(i,k)*((X(j,k)-Cpast(j,i))^2/(Sigmapast(j,i))^3);
                                % Update input mf spreads
     if Sigma(j,i)<sigmabar, Sigma(j,i)=sigmabar;
     end
                             % This makes sure that we do not divide
                             % by zero by making sure that sigma will
                             % not decrease below sigmabar

 end



ave the past values of the parameters for use the next time
round the k loop.

 Bpast=B;
 Cpast=C;
 Sigmapast=Sigma;



:Ngrad*M-1;   % Set time vector for plotting
```

```
Plot the value of e vs. time - just to check afterwards that the
fixed number of steps that we used for training was long enough.

lot(k,e)
label('Time step, k')
itle('Gradient method: error between actual and estimated output')

Next, print out the results of the final step of the gradient
method.

                    %Display output membership function centers
                    %Display intput membership function centers
gma                 %Display the spread for the input membership functions


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
est the fuzzy system at training data pairs and three other
est points.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

t=[Xsave];
ual_output=[Ysave];

mma=501;  % MGamma is the number of test points


 l=1:MGamma

ext compute the fuzzy system - use the same approach as above
y computing the xi vector then the predicted output.

 i=1:R
 muf(i,l)=1;  % Initialization
 numf(i,l)=0;

  for j=1:n
     muf(i,l)=muf(i,l)*exp(-.5*((Xhat(j,l)-C(j,i))/Sigma(j,i))^2);
  end


umoratorf(l)=0;
enominatorf(l)=0;
or i=1:R
 b=Bpast(i);
 numoratorf(l)=numoratorf(l)+muf(i,l)*b;
 denominatorf(l)=denominatorf(l)+muf(i,l);
nd
redicted_output(l)=numoratorf(l)/denominatorf(l);
ercent_error(l)=abs(((predicted_output(l)-actual_output(l)))/(predicted_output(l)↙
tual_output(l)))*100;
     % Compute the output for

         % the test point
```

```
d

Next, display the inputs and outputs

at                    % Display test set
edicted output        % Display predicted output
tual_output           % Display actual output
rcent_error           % Display percent error between predicted and actual ouput

=[-1750: 100 : 1750 ];
=[-650: 50 : 650 ];
ver_x1=-1750;
per_x1=1750;
ver_x2=-650;
per_x2=650;

 i=1:R
 count=1;
 for t=lower_x1: 100: upper_x1
 mux1(i,count)=exp(-0.5*(((t)-C(1,i))/Sigma(1,i))^2);
 count=1+count;
 end


 i=1:R
 count=1;
 for s=lower_x2: 50: upper_x2
 mux2(i,count)=exp(-0.5*(((s)-C(2,i))/Sigma(2,i))^2);
 count=1+count;
 end



ure(1);
t(x1, mux1(:,:));
bel('input 1, x1');
bel('membership value, mu');
le('Gradient method: rule base membership function for rule base input 1');

ure(2);
t(x2, mux2(:,:));
bel('input 2, x2');
bel('membership value, mu');
le('Gradient method: rule base membership function for rule base input 2');
```

Linear Models

Typically, control engineers begin by developing a mathematical description of the dynamic system that they want to control. The system to be controlled is called a *plant*. As an example of a plant, this section uses the DC motor. This section develops the differential equations that describe the electromechanical properties of a DC motor with an inertial load. It then shows you how to use the Control System Toolbox to build linear models based on these equations.

Linear Model Representations

The Control System Toolbox supports the following model representations:

- State-space models (SS) of the form

  $$\frac{dx}{dt} = Ax + Bu$$
  $$y = Cx + Du$$

  where *A*, *B*, *C*, and *D* are matrices of appropriate dimensions, *x* is the state vector, and *u* and *y* are the input and output vectors.
- Transfer functions (TF), for example,

  $$H(s) = \frac{s+2}{s^2+s+10}$$
- Zero-pole-gain (ZPK) models, for example,

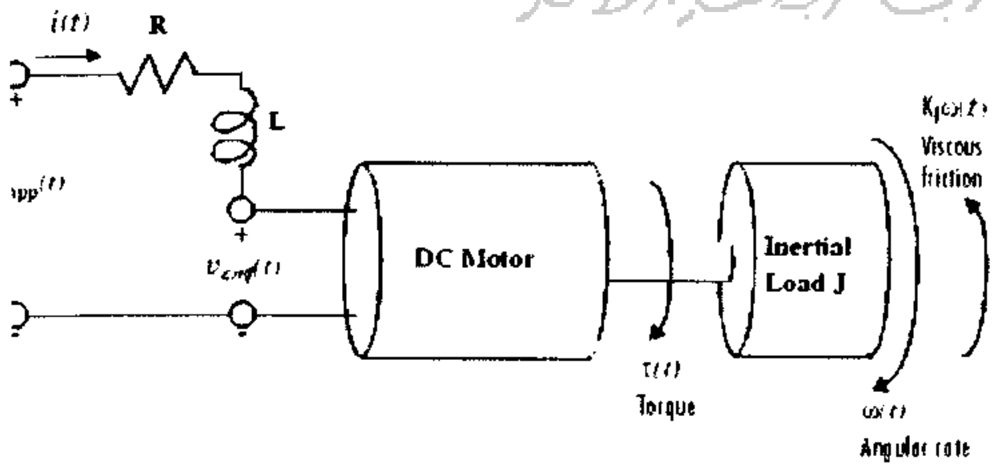
  $$H(z) = 3\frac{(z+1+j)(z+1-j)}{(z+0.2)(z+0.1)}$$
- Frequency response data (FRD) models, which consist of sampled measurements of a system's frequency response. For example, you can store experimentally collected frequency response data in an FRD model.

  Note The design of FRD models is a specialized subject that this guide does not address. See Frequency Response Data (FRD) Models for a discussion of this topic.

SISO Example: The DC Motor

. simple model of a DC motor driving an inertial load shows the angular rate of the load, $\omega(t)$, as the output and pplied voltage, = $v_{app}(t)$, as the input. The ultimate goal of this example is to control the angular rate by varying ıe applied voltage. This picture shows a simple model of the DC motor.

. Simple Model of a DC Motor Driving an Inertial Load



In this model, the dynamics of the motor itself are idealized; for instance, the magnetic field is assumed to be constant. The resistance of the circuit is denoted by R and the self-inductance of the armature by L. If you are unfamiliar with the basics of DC motor modeling, consult any basic text on physical modeling. The important thing here is that with this simple model and basic laws of physics, it is possible to develop differential equations that describe the behavior of this electromechanical system. In this example, the relationships between electric potential and mechanical force are Faraday's law of induction and Ampère's law for the force on a conductor moving through a magnetic field.

Mathematical Derivation

The torque τ seen at the shaft of the motor is proportional to the current i induced by the applied voltage,

$$\tau(t) = K_m i(t)$$

vhere K_m, the armature constant, is related to physical properties of the motor, such as magnetic field strength, the ıumber of turns of wire around the conductor coil, and so on. The back (induced) electromotive force, v_{emf}, is a voltage proportional to the angular rate ω seen at the shaft,

$$v_{emf}(t) = K_b \omega(t)$$

vhere K_b, the emf constant, also depends on certain physical properties of the motor.

'he mechanical part of the motor equations is derived using Newton's law, which states that the inertial load J times he derivative of angular rate equals the sum of all the torques about the motor shaft. The result is this equation,

$$J\frac{d\omega}{dt} = \sum \tau_i = -K_f \omega(t) + K_m i(t)$$

/here $K_f \omega$ is a linear approximation for viscous friction.

'inally, the electrical part of the motor equations can be described by

$$v_{app}(t) - v_{emf}(t) = L\frac{di}{dt} + Ri(t)$$

r, solving for the applied voltage and substituting for the back emf,

$$v_{app}(t) = L\frac{di}{dt} + Ri(t) + K_b \omega(t)$$

'his sequence of equations leads to a set of two differential equations that describe the behavior of the motor, the rst for the induced current,

$$\frac{di}{dt} = -\frac{R}{L} i(t) - \frac{K_b}{L}\omega(t) + \frac{1}{L} v_{app}(t)$$

nd the second for the resulting angular rate,

$$\frac{d\omega}{dt} = -\frac{1}{J} K_f \omega(t) + \frac{1}{J} K_m i(t)$$

tate-Space Equations for the DC Motor

iiven the two differential equations derived in the last section, you can now develop a state-space representation of ıe DC motor as a dynamic system. The current i and the angular rate ω are the two states of the system. The pplied voltage, v_{app}, is the input to the system, and the angular velocity ω is the output.

$$\frac{d}{dt}\begin{bmatrix} i \\ \omega \end{bmatrix} = \begin{bmatrix} -\frac{R}{L} & -\frac{K_b}{L} \\ \frac{K_m}{J} & -\frac{K_f}{J} \end{bmatrix} \begin{bmatrix} i \\ \omega \end{bmatrix} + \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} \cdot v_{app}(t)$$

State-Space Representation of the DC Motor Example

$$y(t) = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ \omega \end{bmatrix} + [0]\ v_{app}(t)$$

Constructing SISO Models

Once you have a set of differential equations that describe your plant, you can construct SISO models using simple ommands in the Control System Toolbox. The following sections discuss

- Constructing a state-space model of the DC motor
- Converting between model representations
- Creating transfer function and zero/pole/gain models

onstructing a State-Space Model of the DC Motor

isted below are nominal values for the various parameters of a DC motor.

```
R= 2.0 % Ohms
L= 0.5 % Henrys
Km = .015 % torque constant
Kb = .015 % emf constant
Kf = 0.2 % Nms
J= 0.02 % kg.m^2
```

iven these values, you can construct the numerical state-space representation using the `ss` function.

```
A = [-R/L -Kb/L; Km/J -Kf/J]
B = [1/L; 0];
C = [0 1];
D = [0];
sys_dc = ss(A,B,C,D)
```

his is the output of the last command.

```
a =
                        x1           x2
           x1           -4        -0.03
           x2         0.75          -10


b =
                        u1
           x1            2
           x2            0


c =
                        x1           x2
           y1            0            1


d =
                        u1
           y1            0
```

onverting Between Model Representations

ow that you have a state-space representation of the DC motor, you can convert to other model representations, cluding transfer function (TF) and zero/pole/gain (ZPK) models.

ransfer Function Representation. You can use `tf` to convert from the state-space representation to the transfer nction. For example, use this code to convert to the transfer function representation of the DC motor.

```
sys_tf = tf(sys_dc)

Transfer function:
        1.5
------------------
s^2 + 14 s + 40.02
```

Zero/Pole/Gain Representation. Similarly, the zpk function converts from state-space or transfer function representations to the zero/pole/gain format. Use this code to convert from the state-space representation to the zero/pole/gain form for the DC motor.

```
sys_zpk = zpk(sys_dc)

Zero/pole/gain:
        1.5
-------------------
(s+4.004) (s+9.996)
```

> **Note** The state-space representation is best suited for numerical computations. For highest accuracy, convert to state space prior to combining models and avoid the transfer function and zero/pole/gain representations, except for model specification and inspection. See Reliable Computations for more information on numerical issues.

onstructing Transfer Function and Zero/Pole/Gain Models

n the DC motor example, the state-space approach produces a set of matrices that represents the model. If you hoose a different approach, you can construct the corresponding models using tf, zpk, ss, or frd.

```
sys = tf(num,den)                    % Transfer function
sys = zpk(z,p,k)                     % Zero/pole/gain
sys = ss(a,b,c,d)                    % State-space
sys = frd(response,frequencies)      % Frequency response data
```

or example, if you want to create the transfer function of the DC motor directly, use these commands.

```
s = tf('s');
sys_tf = 1.5/(s^2+14*s+40.02)
```

he Control System Toolbox builds this transfer function.

```
Transfer function:
        1.5
--------------------
s^2 + 14 s + 40.02
```

lternatively, you can create the transfer function by specifying the numerator and denominator with this code.

```
sys_tf = tf(1.5,[1 14 40.02])

Transfer function:
       1.5
------------------
s^2 + 14 s + 40.02
```

o build the zero/pole/gain model, use this command.

```
sys_zpk = zpk([],[-9.996 -4.004], 1.5)
```

his is the resulting zero/pole/gain representation.

```
Zero/pole/gain:
        1.5
-------------------
(s+9.996) (s+4.004)
```

Discrete Time Systems

The Control System Toolbox provides full support for discrete-time systems. You can create discrete systems in the same way that you create analog systems; the only difference is that you must specify a sample time period for any model you build. For example,

```
sys_disc = tf(1, [1 1], .01);
```

creates a SISO model in the transfer function format.

```
Transfer function:
  1
-----
z + 1

Sampling time: 0.01
```

Adding Time Delays to Discrete-Time Models

You can add time delays to discrete-time models by specifying an input or output time delay when building the model. The time delay must be a nonnegative integer that represents a multiple of the sampling time. For example,

```
sys_delay = tf(1, [1 1], 0.01,'outputdelay',5);
```

produces a system with an output delay of 0.05 second.

```
Transfer function:
           1
z^(-5) * -----
         z + 1

Sampling time: 0.01
```

Adding Delays to Linear Models

You can add time delays to linear models by specifying an input or output delay when building a model. For example, to add an input delay to the DC motor, use this code.

```
sys_tfdelay = tf(1.5, [1 14 40.02],'inputdelay',0.05)
```

The Control System Toolbox constructs the DC motor transfer function, but adds a 0.05 second delay.

```
Transfer function:
                        1.5
exp(-0.05*s) * ------------------
               s^2 + 14 s + 40.02
```

For a complete description of adding time delays to models, see Time Delays.

LTI Objects

For convenience, the Control System Toolbox uses custom data structures called *LTI objects* to store model-related data. For example, the variable sys_dc created for the DC motor example is called an *SS object*. There are also TF, ZPK, and FRD objects for transfer function, zero/pole/gain, and frequency data response models respectively. The four LTI objects encapsulate the model data and enable you to manipulate linear systems as single entities rather than as collections of vectors or matrices.

To see what LTI objects contain, use the get command. This code describes the contents of sys_dc from the DC motor example.

```
get(sys_dc)
                a: [2x2 double]
                b: [2x1 double]
                c: [0 1]
                d: 0
                e: []
        StateName: {2x1 cell}
    InternalDelay: [0x1 double]
               Ts: 0
       InputDelay: 0
      OutputDelay: 0
        InputName: {''}
       OutputName: {''}
       InputGroup: [1x1 struct]
      OutputGroup: [1x1 struct]
             Name: ''
            Notes: {}
         UserData: [] OutputName: {''}
     InputGroup: {0x2 cell}
    OutputGroup: {0x2 cell}
          Notes: {}
       UserData: []
```

You can manipulate the data contained in LTI objects using the set command; see the Control System Toolbox online reference pages for descriptions of set and get.

Another convenient way to set or retrieve LTI model properties is to access them directly using dot notation. For example, if you want to access the value of the A matrix, instead of using get, you can type

```
    sys_dc.a
```

at the MATLAB prompt. MATLAB returns the A matrix.

```
    ans =

       -4.0000   -0.0300
        0.7500  -10.0000
```

Similarly, if you want to change the values of the A matrix, you can do so directly, as this code shows.

```
    A_new = [-4.5 -0.05; 0.8 -12.0];
    sys_dc.a = A_new;
```

For more information on LTI properties, type ltiprops at the MATLAB prompt. For a complete description of LTI objects, see Creating and Manipulating Models.

Building Models   MIMO Models