MATLAB and Simulink Support

Appendix

A

MATLAB, an abbreviation for MATrix LABoratory, is a high-performance language for technical computing. It integrates computation, visualization, and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation.

The MATLAB family of programs includes the *base program* plus a variety of application-specific solutions called *toolboxes*. Toolboxes are comprehensive collections of MATLAB functions that extend the MATLAB environment to solve particular classes of problems. Together, the base program plus the *Control System Toolbox* provide the capability to use MATLAB for control system analysis and design problems at the level of the text. Whenever MATLAB is referred to in this book, you can interpret that to mean the base program plus the control system toolbox.

Simulink, a companion program to MATLAB, is an interactive system for simulating nonlinear dynamic systems. It is a graphical mouse-driven program that allows you to model a system by drawing a block diagram on the screen and manipulating it dynamically. It is a powerful, comprehensive, and user-friendly software package for simulation studies.

If you are new to MATLAB and Simulink, and want to learn it quickly, start by visiting the web course: MATLAB Modules for Control System Principles and Design [151]. The course has been designed to serve the purpose of giving a 'starting kick' to the students who need to be kicked to action. The course has not been designed as a substitute to the MATLAB manuals. MathWorks [152] provides extensive documentation in both printed and online format to help you learn about and use all the features of MATLAB and Simulink. The online **help** provides task-oriented, and function-reference information. The documentation is also available in PDF format. Visit the website of MathWorks [152].

Our presentation in this appendix has a limited objective of helping the reader refresh his/her knowledge on MATLAB and Simulink. We use MATLAB version 7. For the most part, the material applies to 6x version as well. MATLAB continues to evolve as a software tool. However, the basic operation of MATLAB and its basic capabilities have, more or less, stabilized around the optimum level. Dramatic changes are not expected in near future; the material, therefore, will continue to apply to 7x version as well.

A.1 MATLAB FUNCTIONS FOR CONTROL DESIGN

A.1.1 Models

When a control engineer is given a control problem, often one of the first tasks that he undertakes is the development of the mathematical model of the process to be controlled. We can use first principles of physics to write down a model. Another way is to perform "system identification" via the use of real plant data to produce a model of the system.

For linear time-invariant systems, mathematical model building based on physical laws normally results in a set of (first-order and second-order) differential equations. These equations, when rearranged as a set of first-order differential equations, result in a state-space model of the following form for single-input, single-output (SISO) systems:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{b}u \\ y &= \mathbf{c}\mathbf{x} + du \end{aligned} \tag{A.1}$$

where $\mathbf{x}(t)$ is the $n \times 1$ internal state vector, $u(t)$ is the control input and $y(t)$ is the measured output. Overdot represents differentiation with respect to time t.

Complex processes and machines often have several manipulated inputs available to provide this control (multivariable or MIMO system). In many situations, one input affects primarily one output and has only weak effect on other outputs; it becomes possible to ignore weak interactions (coupling) and design controllers under the assumption that one input affects only one output. Input-output pairing to minimize the effect of interactions and application of SISO control schemes to obtain separate controllers for each input-output pair, results in an acceptable performance. This, in fact, amounts to considering the MIMO system as consisting of an appropriate number of separate SISO systems. Coupling effects are considered as disturbances to the separate control systems. Though MATLAB has a provision for both the SISO and MIMO models, we will limit our discussion to SISO models.

For the state-space representation (A.1), the data for the model consists of four matrices. For convenience, the MATLAB provides customized data structure. This is called the **SS** object. This object encapsulates the model data and enables you to manipulate linear time-invariant (LTI) system as single entity, rather than collection of data vectors and matrices.

An LTI object of the type **SS** is created whenever you invoke the construction function **ss**. For example, type

```
>> A = [– 8   – 16   – 6;   1   0   0 ; 0   1   0];
>> b = [1;   0;   0];
>> c = [2   8   6];
>> d = 0;
>> sys = ss(A, b, c, d)
```

and MATLAB responds with

```
a =
                x1          x2          x3
    x1          – 8         – 16        – 6
    x2          1           0           0
    x3          0           1           0
b =
                u1
    x1          1
    x2          0
    x3          0
c =
                x1          x2          x3
    y1          2           8           6
d =
                u1
    y1          0
```

Controllability and observability of a system in state variable form can be checked using the MATLAB functions **ctrb** and **obsv**, respectively. The inputs to the **ctrb** function are the system matrix **A** and the input matrix **b**; the output of **ctrb** function is the controllability matrix **U**. Similarly, the inputs to the **obsv** function are the system matrix **A** and the output matric **c**; the output is observability matrix **V**. The functions **det(U)/rank(U)** and **det(V)/rank(V)** give, respectively, the controllability and observability properties.

Our next step is to make a design model. Since our focus is going to be on frequency-domain design methods, a transfer function model will be required. A transfer function

$$G(s) = \frac{n(s)}{d(s)} \tag{A.2}$$

is characterized by the numerator $n(s)$ and denominator $d(s)$; both polynomials of the Laplace variable s. MATLAB provides LTI object **TF** for transfer functions. The object **TF** is created whenever you invoke the construction function **tf**. For example, for

$$G(s) = (8 + 12s + 4s^3)/(17s^5 + 23s^3 + 8s^2 + 6), \text{ type}$$

```
>> num = [4   0   12   8]; den = [17   0   23   8   0   6];
>> sys = tf(num, den)
```

and MATLAB responds with

Transfer function

$$\frac{\mathbf{4s\ ^\ 3 + 12s + 8}}{\mathbf{17s\ ^\ 5 + 23s\ ^\ 3 + 8s\ ^\ 2 + 6}}$$

Quite often, the transfer function in hand is in zero-pole-gain form,

$$G(s) = \frac{10(s-3)(s+4.5)(s+100)}{(s+40)^3(s+80)}$$

MATLAB provides construction function **zpk** to create **ZPK** object. We can, however, create **TF** object for this transfer function using construction function **tf**.

```
>> r1 = [3; – 4.5; – 100]; r2 = [– 40; – 40; – 40; – 80];
>> num = 10* poly(r1); den = poly(r2);
>> sys = tf(num, den);
```

r1 is a column vector containing the roots of a polynmial. The function **poly(r1)** assembles the polynomial.

Some more examples of construction of **TF** object follow.

$$G(s) = \frac{10(s+2)(s+5)}{(s^2+2s+5)(s+3)}$$

with real as well as complex poles.

```
>> r = [– 2; – 5];
>> num = 10 * poly(r);
>> p = [1 2 5];  q = [1   3];
>> den = conv(p, q);
>> sys = tf(num, den);
```

The function **conv** has been used to multiply two polynomials $p(s) = s^2 + 2s + 5$, and $q(s) = s + 3$.

$$G(s) = \frac{100(5s+1)(15s+1)}{s(3s+1)(10s+1)}$$

is a transfer function in time-constant form.

```
>> n1 = [5   1]; n2 = [15   1];   d1 = [1   0]; d2 = [3   1];
>> d3 = [10   1]; num = 100 * conv(n1, n2);
>> den = conv(d1, conv(d2, d3));
>> sys = tf(num, den);
```

To create the object **TF** directly, use these commands:

```
>> s = tf('s');
>> sys = 100*(5*s+1)*(15*s+1)/[s*(3*s+1)*(10*s+1)];
```

In cases where our knowledge of the system under study is limited, or the theoretical model turns out to be highly complex, the only reliable information on which to base the control design is the experimental data. MATLAB has a provision of creating an LTI object called **FRD** which stores frequency response data (complex frequency response, alongwith a corresponding vector of frequency points) that you obtain experimentally.

MATLAB has the means to perform model conversions. Given the **SS** model **sys_ss**, the syntax for conversion to **TF** model is

```
sys_tf = tf(sys_ss)
```

Common pole-zero factors of the **TF** model must be cancelled before we can claim that we have the transfer function representation of the system. To assist us in pole-zero cancellation, MATLAB provides **minreal** function.

```
sysr = minreal(sys_tf)
```

Given the **TF** model **sys_tf**, the syntax for conversion to **SS** model is

```
sys_ss = ss(sys_tf)
```

Process transfer function models frequently have dead-time (input-output delay). **TF** object for transfer functions with dead-time can be created using the syntax

```
sys = tf(num, den, 'InputDelay', value)
```

For $G(s) = (6.6\, e^{-7s})/(10.9s + 1)$, type

```
>> num = [6.6]; den = [10.9   1];
>> sys = tf(num, den, 'InputDelay', 7);
```

Two cascaded blocks with transfer functions $G_1(s)$ and $G_2(s)$ can be multiplied using the **series** function.

```
sys1 = tf(num1, den1) % G1(s)
sys2 = tf(num2, den2) % G2(s)
 sys = series(sys1, sys2) % G1(s)G2(s)
```

or

```
 sys = sys1 *sys2
```

If **sys1** and **sys2** are in parallel, then

```
 sys = parallel(sys1, sys2) % G1(s) + G2(s)
```

or

```
 sys = sys1 + sys2
```

For a negative feedback loop with forward path transfer function $G(s)$ and feedback path transfer function $H(s)$, the function **feedback** results in closed-loop transfer function.

```
sys1 = tf(num1, den1) % G(s)
sys2 = tf(num2, den2) % H(s)
 sys = feedback(sys1, sys2) % G(s)/(1 + G(s)H(s))
```

For a unity-feedback system, the closed-loop transfer function is given by

```
sys = feedback(sys1, 1) % G(s)/(1 + G(s))
```

A.1.2 Time Response

For the **SS** object of the model (A.1):

```
sys = ss(A, b, c, d)
```

the function **step(sys)** will generate a plot of unit-step response $y(t)$ (with zero initial conditions). The time vector is automatically selected when **t** is not explicitly included in the **step** command.

If you wish to supply the time vector **t** at which the response will be computed, the following command is used.

```
step(sys, t)
```

You can specify either a final time **t = Tfinal,** or a vector of evenly spaced time samples of the form

```
t = 0 : dt : Tfinal
```

When invoked with left-hand arguments such as

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

no plot is generated on the screen. Hence it is necessary to use a **plot** command to see the response curves. The vector **y** and matrix **X** contain the output and state response of the system, respectively, evaluated at the computation points returned in the time vector **t** (**X** has as many columns as states and one row for each element in vector **t**).

Other time-response functions of interest to us are:

```
impulse(sys)        % impulse response
initial(sys,x0)     % free response to initial state vector x0
lsim(sys,u,t)       % response to input time history
lsim(sys,u,t,x0)    % in vector u having length (t) rows
```

For the **TF** object of the model (A.2):

```
sys = tf(num, den)
```

the function **step(sys)** will generate a unit-step response $y(t)$. The time vector is automatically selected when **t** is not explicitly included in the **step** command. If you wish to supply the time vector **t** at which the response will be computed, the following command is used.

```
step (sys,t)
```

When invoked with left-hand arguments such as

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

no plot is generated on the screen. Hence it is necessary to use a **plot** command to see the response curve. The vector **y** has one column and one row for each element in time vector **t**.

Other time-response functions of interest to us are:

```
impulse(sys)  % impulse response
lsim(sys,u,t)  % response to input time history in vector u having
               % length (t) rows.
```

Place your mouse on any point along a plot line (step, impulse, lsim, initial). Left-clicking on this point opens data with relevant information displayed.

As the nature of the transient response of a control system is dependent upon the system poles only and not on the type of the input, it is sufficient to analyze the transient response to one of the standard test signals; a step is generally used. In specifying the steady-state response characteristics, it is common to specify the steady-state error of the system to one or more of the standard test signals—step, ramp, parabola.

Typical design specifications demand that the system have (when subjected to command/disturbance inputs)

(i) a step response inside some constraint boundaries—specified by settling time, peak overshoot, etc; and

(ii) steady-state error to step/ramp/parabolic input within prescribed limits, under the constraints imposed by physical limitations of the selected plant, actuator, and sensor.

You can analyze the time response using the **GUI** (graphical user interface) for viewing and manipulating response plots of LTI models. For example, **step (sys)** will open a window displaying the step response of the LTI model **sys**. Once initialized, the **GUI** assists you with the analysis of the response by facilitating such functions as zooming into regions of the response plots; calculating response characteristics such as peak response, settling time, rise time, steady-state; toggling the grid on or off the plot; and many other useful features.

Right-click anywhere in the plot region of the step response plot. This opens a menu list in the plot region. Select the **Grid** menu item with left mouse button. A grid for the graph appears. Now right-click again and place the pointer in the **Characteristics** menu item. The submenu items (Peak Response, Settling Time, Rise Time, Steady-State) of the Characteristics menu are displayed. Select **Settling Time** with left mouse button. The settling time marker appears on the graph. Place your mouse on the marker. This opens data with relevant information displayed. To make it persistent, left-click on the marker.

In addition to right-click menus, the GUI provides plot-data markers. Left-click anywhere on a plot line; a data marker appears with the response values of the plot at that point displayed. You can move the data marker along the plot line. Move the mouse pointer over the marker. The pointer becomes a hand. Grab the marker by holding down the left mouse button when the hand appears. Drag the marker with your mouse and release the mouse button at a point of your interest along the plot line. Response values at the selected point are displayed.

You can use the **Property Editor** to customize various attributes of your plot. Move the pointer on the **Properties** menu item to open the editor.

The GUI feature is available with all the time response plots.

A.1.3 Frequency Response

For the **TF** object of the model (A.2):

sys = tf(num, den)

the function **bode(sys)** generates the Bode frequency-response plots. This function automatically selects the frequency values by placing more points in regions where the frequency response is changing quickly. This range is user-selectable using the command **bode(sys,w)**. Since the Bode diagram has log scale, if we choose to specify the frequencies explicitly, it is desirable to generate the vector **w** using the **logspace** function. When invoked with left-hand arguments,

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

[mag,phase] = bod(sys,w)

return the magnitude and phase of frequency response at the frequencies **w**.

nyquist(sys) plots the real and imaginary parts of the frequency response of an arbitrary LTI model **sys**. **nyquist (sys,w)** explicitly specifies the frequency range to be used for the plot. To focus on a particular frequency interval, set **w = {wmin,wmax}**. When invoked with left-hand arguments,

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

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

return the real and imaginary parts of the frequency response at the frequencies **w**.

Sometimes in the course of using the **nyquist** function, we may find that a Nyquist plot looks nontraditional or that some information appears to be missing. It may be necessary, in these cases, to override the automatic scaling and focus in on the $-1 + j0$ point region for stability analysis.

In practice, Nyquist diagrams are commonly plotted on the Nichols coordinate system with rectangular coordinate axes for the phase (in degrees) and the gain (in dB). On the Nichols coordinate system, the critical point for stability becomes (180°, 0dB).

The Nyquist diagram on the Nichols coordinate system can be generated using the **nichols** function. **nichols(sys)** produces a Nyquist plot of the LTI model **sys**. **nichols(sys,w)** explicitly specifies the frequency range to be used for the plot. When invoked with left-hand arguments,

[mag,phase,w] = nichols(sys)

[mag,phase] = nichols(sys,w)

return the magnitude (in dB) and phase (in degrees) of the frequency response at the frequencies **w** (in rad/sec). Nichols chart grid is drawn on the existing plot using **ngrid** function.

Requirements that a control system have a step response inside some constraint boundaries—specified by settling time, peak overshot, etc., can equivalently be represented as requirements that the system have a frequency response satisfying certain constraints specified by gain margin, phase margin, bandwidth, etc. The function **margin** determines gain margin, phase margin, gain crossover frequency, and phase crossover frequency. Resonance peak, resonance frequency, and bandwidth of a closed-loop system may be obtained from the plot generated by **nichols** and **ngrid** functions.

You can analyze the frequency response using the **GUI** (graphical user interface) for viewing and manipulating response plots of LTI models. For example, **nichols (sys)** will open a window displaying the Nichols plot of the LTI system **sys**. Once initialized, the **GUI** assists you with the analysis of the response by facilitating such functions as zooming into regions of response plot; calculating response characteristics such as resonance peak, resonance frequency, bandwidth, stability margins; and many other useful features.

Right-click anywhere in the plot region of the Nichols plot. This opens a menu list in the plot region. Select the **Grid** menu item with left mouse button. A grid for the graph appears. Identify the –3 dB contour on the grid. Zoom in on the region of intersection of this contour with the plot. Click on the point of intersection; hold the mouse button down to read the value of bandwidth.

In addition to right-click menus, the GUI provides plot-data markers. Left-click anywhere on a plot line; a data marker appears with the response values of the plot at that point displayed. You can move the data marker along the plot line. Move the mouse pointer over the marker. The pointer becomes a hand. Grab the marker by holding down the left mouse button when the hand appears. Drag the marker with your mouse and release the mouse button at a point of your interest along the plot line. Response values at the selected point are displayed.

You can use the **Property Editor** to customize various attributes of your plot. Move the pointer on the **Properties** menu item to open the editor.

The GUI feature is available with all the frequency-response plots.

A.1.4 Design

Root Locus: If **sys** models a transfer function

$$G(s) = \frac{n(s)}{d(s)}$$

rlocus adaptively selects a set of positive gains K and produces a smooth plot of the roots of

$$d(s) + Kn(s) = 0$$

Alternatively, **rlocus(sys,K)** uses the user-specified vector **K** of gains to plot the root locus. Left-click anywhere on the root loci to see the relevant information about the locus at that point.

The function **rlocfind** returns the feedback gain associated with a particular set of poles on the root locus. **[K,poles] = rlocfind(sys)** is used for interactive gain selection. The function **rlocfind** puts up a cross hair cursor on the root locus plot that you use to select a particular pole location. The root locus gain associated with this point is returned in **K** and the column vector **poles** contains the closed-loop poles for this gain. To use this command, the root locus must be present in the current figure window.

The function **sgrid/spchart** is used for ω_n and ζ grid on the root locus.

Note that LTI functions are available in figure windows of the commands **margin**, **bode**, **nyquist**, **nichols**, **rlocus**. Left-click anywhere on a particular plot line to see the response values of that plot at that point. You can drag the data marker along a plot line. Also right-click menus are available.

Lag/Lead Design: To form an initial estimate of the complexity of the design problem, sketch frequency response (Bode plot) and root-locus plot with respect to plant gain. Try to meet the specifications with a simple controller of lag/lead variety. Compare a lead network in the forward path to minor-loop feedback structure having direct feedback from velocity sensor, to see which gives a better design.

For design by root-locus method, the design specifications are translated into desired dominant closed-loop poles. Other closed-loop poles are required to be located at a large distance from the $j\omega$-axis. It may be noted that pole-placement methods do not allow the designer to judge how close the system performance is to the best possible. Also, there is lack of visibility into low-frequency disturbance rejection. This can cause many problems: the disturbance rejection may not be optimized, and the plant-parameter variations may cause large closed-loop response variations. Stability margins on Nyquist/Bode plot give a better robustness measure. For this reason, though the specifications on the closed-loop performance are often formulated in time domain, it is worthwhile to convert them into frequency-domain specifications and then design the compensator with frequency-domain methods. Root-locus plots are very impressive analysis tools for systems that have been designed by frequency-domain methods. For example, these plots can be valuable for the analysis of the effects of certain parameter variations on stability.

You can design a compensator using frequency-response MATLAB functions **bode** and **nichols**.

Suppose analog compensator $D(s)$ has been created in MATLAB as **sysc**. The command

```
sysd = c2d(sysc,T,'tustin') % Tustin approximation
                            % T is sampling interval
```

converts the continuous-time system $D(s)$ to discrete-time system using trapezoidal rule for integration. The object **sysc** may be an **SS** object or a **TF** object.

A.1.5 Simulation of Performance of Design

After reaching the best compromise among process modification, actuator and sensor selection, and controller design choice, run a computer model of the feedback system. This model should include important nonlinearities—such as actuator saturation, and parameter variations you expect to find during operation of the system. The simulation will confirm stability and robustness, and allow you to predict the true performance you can expect from the system.

MATLAB provides many Runge Kutta numerical integration routines for solving ordinary differential equations; the function **ode23** usually suffices for our applications. The feedback system is represented by a set of linear/nonlinear state-space equations; the system is coded in an **M-file**, and then **ode** solver function such as **ode23** is applied to solve the system on a given time interval with a particular initial condition vector. Refer [155] for details.

MATLAB's Simulink is better suited for simulation studies in our applications.

More on Design and Performance Analysis: In Chapters 11-14, we have given a brief account of a range of topics on digital control, state variable methods, and nonlinear systems. Detailed account of these topics is available in the companion book [155]. Also included in the companion book is an appendix on MATLAB and Simulink support on these topics. This appendix is available at: http://www.mhhe.com/gopal/dc3e

A.2 MATLAB/Simulink

In the simulation process, the computer is provided with appropriate input data and other information about system structure, operates on this input data and generates output data, which it subsequently displays. Several software packages that have been produced over the last two decades include computer programs that allow these simulation operations. Over the years, these simulation packages have become quite sophisticated, powerful and very "user-friendly". The usefulness and importance of these software packages is undeniable, because they greatly facilitate the analysis and design of control systems. They provide a tremendous tool in the hands of control engineers.

MATLAB/Simulink is one of the most successful software packages currently available, and is particularly suited for work in control. It is a powerful, comprehensive and user-friendly software package for simulation studies. Our objective here is to help the reader gain a basic understanding of this software package by showing how to set up and solve a simulation problem. Interested readers are encouraged to further explore this very complete and versatile mathematical computation package [151, 152].

A very nice feature of Simulink is that it visually represents the simulation process by using *simulation block diagrams*. Especially, functions are represented by "subsystem blocks" that are then interconnected to form a Simulink block diagram that defines the system structure. Once the structure is defined, parameters are entered in the individual subsystem blocks that correspond to the given system data. Some additional simulation parameters must also be set to

govern how the numerical computation will be carried out and how the output data will be displayed. As a matter of fact, the Simulink block diagrams are essentially the same we have used in the text to describe control system structures and signal flow.

Because Simulink is graphical and interactive, we encourage you to jump right in and try it. To help you start using Simulink quickly, we describe here the simulation process though a demonstration example on Microsoft Windows platform with MATLAB version 7, Control Toolbox version 6.1 and Simulink version 6.1.

To start Simulink, enter **simulink** command at the MATLAB prompt. Simulink Library Browser appears which displays tree-structured view of the Simulink block libraries. It contains several nodes; each of these nodes represents a library of subsystem blocks that is used to construct simulation block diagrams. You can expand/collapse the tree by clicking on the ⊞/⊟ boxes beside each node, and block in the block set pan.

Expand the node labeled "Simulink". Subnodes of this node are displayed. Expanding the "Sources" subnode displays a long list of Sources library blocks; contents are displayed in the diagram view. The purpose of the block "Step" is to generate a step function. The block "Constant" generates a specified real or complex value, independent of time. Simply click on any block to learn about its functionality in the description box.

You may now collapse the Sources subnode, and expand the "Sinks" subnode. A list of Sinks library blocks appears. The purpose of block labeled "XY Graph" is to display an X-Y plot of signals using a MATLAB figure window. The block has two scalar inputs; it plots data in the first input (the x direction) against data in the second input (the y direction). The block "Scope" displays its inputs (signals generated during a simulation) with respect to simulation time. The block "To Workspace" transfers the data to MATLAB workspace.

You may now collapse the Sinks subnode and expand the "Continuous" subnode. A list of library blocks corresponding to this subnode appears. The purpose of the "Derivative" block is to output the time derivative of the input. The "State-Space" block implements a linear system whose behaviour is described by a state variable model. The "Transfer Fcn" block implements a transfer function.

The "Discontinuities" subnode has blockset of various nonlinearities: Backlash, Coulomb and Viscous Friction, Deadzone, Relay, Saturation, etc.

The "Math Operations" subnode has several blocks. The block "Sum" generates the sum of inputs. It is useful as an error detector for control system simulations. The "Sign" block indicates the sign of the input (The output is 1 when the input is greater than zero: the output is 0 when the input is equal to zero; and the output is -1 when the input is less than zero).

Expand now the node "Control System Toolbox". The block "LTI system" accepts the continuous and discrete objects as defined in the Control System Toolbox. Transfer functions and state-space formats are supported in this block.

We have described some of the subsystem libraries available that contain the basic building blocks of simulation diagrams. The reader is encouraged to explore the other libraries as well. You can also customize and create your own blocks. For information on creating your own blocks, see the MATLAB documentation on "Writing S-Functions".

We are now ready to proceed to the next step, which is the construction of a simulation diagram. To do this, we need to open a new window. Click the **New** button on the Library Browser's toolbar. A new window opens up that will be used to build up an interconnection of Simulink blocks from the subsystem libraries. This is an "untitled" window; we call it the Simulation Window. We consider here analysis of electromechanical servo system described in Review Example 10.4 (Fig. 10.34).

With the "Sources" subnode of Simulink node expanded, move the pointer and click the block labeled "Step"; while keeping the mouse button pressed down, drag the block and place it inside the Simulation Window, and release the mouse button. We will use "Step" for command inputs. Duplicate "Step" for disturbance inputs.

You can duplicate blocks in a model as follows. While holding down the **Ctrl** key, select the block with the mouse button, then drag it to the new location and release the mouse button.

Drag to the Simulation Window the block labeled "Sum" from the "Math Operations" subnode of Simulink node. Make two more copies of the "Sum" block. Now drag the block labeled "Gain" from "Math Operations" subnode. Make three more copies of this block.

With the "Continuous" subnode expanded, click the block labeled "Transfer Fcn", and drag it to the Simulation Window. Duplicate "Transfer Fcn". Also drag to the Simulation Window "Integrator" block from the "Continuous" subnode.

Drag the blocks "To Workspace" and "Scope" from "Sinks" subnode.

We have now completed the process of dragging subsystem blocks from the appropriate libraries and placing them in the Simulation Window. The next step is to interconnect these subsystem blocks and obtain the structure of simulation block diagram. To do this, we just need to work in the Simulation Window.

The first step is to rearrange the blocks in the Simulation Window in a specified structure. This will require moving a block from one place to another within the Simulation Window. This can be done by clicking inside the block, keeping the mouse button pressed, dragging the block to the new desired location and releasing mouse button.

Select "Gain" block in the feedback path. Press (**Ctrl+R**) to rotate it clockwise by 90°. Repeat rotate step to obtain 180° rotation of this "Gain" block.

Lines are drawn to interconnect these blocks as per the desired structure. A line can connect the output port of one block with the input port of another block. A line can also connect the output port of one block with input ports of many blocks by using branch lines.

To connect the output port of one block to the input port of another block, position the pointer on the first block's output port; the pointer shape changes to a crosshair. Press and hold down the mouse button. Drag the pointer to the second block's input port. You can position the pointer on or near the port; the pointer shape changes to a double crosshair. Release the mouse button. Simulink replaces the port symbols by a connecting line with an arrow showing the direction of signal flow.

A *branch line* is a line that starts from an existing line and carries its signal to the input port of a block. Both the existing line and the branch line carry the same signal. To add a branch line, position the pointer on the line where you want the branch line to start. While holding down the **Ctrl** key, press and hold down the mouse button. Drag the pointer to the input port of the target block, then release the mouse button and the **Ctrl** key.

The branch lines are usually an interconnection of line segments. With the **Ctrl** key pressed, identify the branch point and drag the mouse (horizontally/vertically) to an unoccupied area of the diagram and release the mouse button. An arrow appears on the unconnected end of the line. To add another line segment, position the pointer over the end of the segment and draw another segment.

To move a line segment, position the pointer on the segment you want to move. Press and hold down the left mouse button. Drag the pointer to the desired location and release.

To disconnect a block from its connecting lines, hold down the **Shift** key, then drag the block to a new location. You can insert a block in a line by dropping the block on the line.

You can cancel the effects of an operation by choosing **Undo** from the **Edit** menu of the Simulation Window. You can thus undo the operations of adding/deleting a line/block. Effects of **Undo** command may be reversed by choosing **Redo** from the **Edit** menu.

To delete a block/line, select a block/line to be deleted and choose **Clear** or **Cut** from the **Edit** menu. The **Cut** command writes the block/line into the clipboard, which enables you to **Paste** it into a model. **Clear** command does not enable you to paste the block/line later.

All block names in a model must be unique and must contain at least one character. By default, block names appear below blocks. To edit a block name, click on the block name and insert/delete/write text. After you are done, click the pointer somewhere else in the model, the name is accepted or rejected. If you try to change the name of a block to a name that already exists, Simulink displays an error message.

This gives us a generic diagram because we have not yet specified the parameter values of the blocks. Our next priority is to go into each of these blocks and set the parameters that correspond to our specific system. In addition, we need to set some simulation parameters.

We begin with the command input to the feedback system by double-clicking on the block labeled "Command" in the Simulation Window. A dialog box pops up. We set the parameters: Step time 0, Initial value 0, and Final value 1. The parameter values of the "Disturbance" block are all set to 0 initially, i.e, we set the parameters to obtain the response to command inputs only.

Next we set the "Sum" block. In the dialog box for this block we enter Icon shape: round, and list of signs, + –. This gives us an error detector for negative feedback system.

We set the "Amp gain" to 4.1545, "Torque const" to 0.0952, "Gear ratio" to 1/10.5, and "Back emf const" 0.0952. The "Armature" circuit transfer function is set to $1/(0.01s + 3.086)$, and the "Load" when reflected to motor shaft is represented by $1/(0.000041804s + 0.000475)$.

"To Workspace" block requires variable name and the format. We use **Array** format for our data and enter variable name **Response** in the dialog box.

Finally, we need to set the parameters for the simulation run. We move the pointer to the menu labeled "Simulation", and enter Configuration Parameters: start time, stop time, in the dialog box.

At this point in the simulation process, we have generated the appropriate Simulink block diagram (shown in Fig. A.1) and entered the specific parameters for our system and simulation. We are now ready to execute the program, and have the computer perform the simulation. We move the pointer to the "Simulation" menu and choose "Start". Double-click the "Scope" block to view the simulation result (click "autoscale" on toolbar if required).

You may now execute the following program in MATLAB workspace.

```
figure (1);
plot (tout, Response); grid;
hold on
```

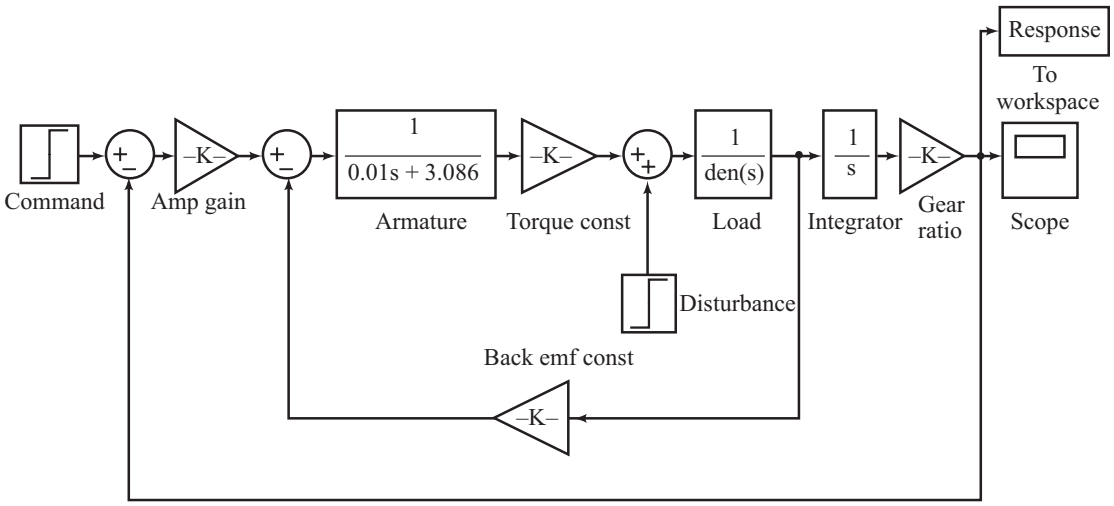

Fig. A.1

We have used an example to show how to enter data and carry out a simulation in the Simulink environment. The reader will agree that this is a very simple process. Download the file **SimulinkFigA1** in MATLAB environment. Double-click each block and study the properties of the block (You may change these properties as per your analysis requirement).

Study/modify simulation parameters, and execute the program.

Problems

Each problem covers an important area of control-system analysis or design. Important MATLAB commands have been included as help to these problems, in the form of scrip files.

Open these files in MATLAB environment. In attempting a problem, the reader can use the MATLAB commands in the script file in an interactive manner, or use the script file as an **M-file**. The description of the MATLAB functions in the Script files can easily be accessed from the **help file** using **help** command.

Simulink files are included as help to some problems. Download the Simulink files from the URL. Open these files in MATLAB environment. Doube-click and study the properties of each block.

Following each problem, one or more what-if's may be posed to examine the effect of variations of the key parameters. Comments to alert the reader to the special features of MATLAB commands are included in the script files to enhance the learning experience. Partial answers to the problems are also included.

A.1 Figure 4.6 shows the model of a heat exchanger control loop. Plot response of the system to unit-step input θ_r, for $K_A = 10$, and for $K_A = 5$.
Comment on the effect of amplifier gain on transient and steady-state accuracy.

A.2 Figure 4.6 shows the model of a heat exchanger control loop. Plot response of the system to unit-step disturbance θ_i, for $K_A = 10$, and for $K_A = 5$.
Comment on the effect of amplifier gain on transient and steady-state accuracy.

A.3 Figure 4.6 shows the model of a heat exchanger control loop. Determine the closed-loop poles of the system. Sketch a pole-zero map and therefrom comment upon stability.

A.4 *Review Example 5.3 revisited.*
A non-unity feedback system has process transfer function

$$G(s) = \frac{K}{s(s+1)}$$

and feedback transfer function

$$H(s) = \frac{1 - Ts}{1 + Ts}$$

What are the combinations of K and T for which the system is stable?

A.5 *The Script PA.4 revisited.*
The MATLAB response to this script shows the stability region for a non-unity feedback system. Take a point in this region and determine the response of the resulting feedback system to a unit-ramp input.

A.6 *Review Example 6.2 revisited.*
A unity-feedback system is characterized by the open-loop transfer function

$$G(s) = \frac{1}{s(0.5s+1)(0.2s+1)}$$

(a) Determine the damping ratio and natural frequency of dominant closed-loop poles.
(b) Determine the error constants K_p, K_v, and K_a.
(c) Determine peak overshoot, time to peak, and settling time of the step response of the feedback system. Are your results different from the ones given in the text? Why?

(Ans: $M_p = 12.1186\%$; $t_p = 3$ sec; $t_s = 4.52$ sec)

A.7 *Examples 6.2 and 6.4 revisited.*
A unity-feedback position control system has open-loop transfer function

$$G(s) = \frac{4500}{s(s+361.2)}$$

Plot step response of the system with a cascade controller $D(s)$ with
(a) $D(s) = 184.1$
(b) $D(s) = 184.1 + 0.324s$
(c) $D(s) = 14.728 + 147.28/s$
Compare the responses for (a) and (b), and those for (a) and (c). Comment upon the effects of derivative and integral control actions.

A.8 *Examples 9.2 and 9.4 revisited.*
A unity-feedback system has open-loop transfer function

$$G(s) = \frac{10}{s(1+0.1s)(1+0.05s)}$$

(a) Determine gain margin, phase margin, gain crossover frequency and phase crossover frequency.
(b) Determine resonance peak, resonance frequency, and bandwidth.
The answers will be slightly different from the ones given in the text. The text answers are based upon Bode plot obtained using asymptotic approximation.

A.9 *Design Example 1 in Section 7.5 revisited.*
A unity-feedback system has open-loop transfer function

$$G(s) = \frac{K}{s(s+1)(s+5)}$$

Relative stability specification calls for a peak overshoot of 14%. Find the value of K to meet this specification and the resulting closed-loop poles. What is the natural frequency of dominant pair of poles?
Using GUI, determine the peak overshoot and settling time of the closed-loop step response.
(Ans: 19%; 9.51 sec)

A.10 *Example 7.8 revisited.*
A unity-feedback system has open-loop transfer function

$$G(s) = \frac{K}{s(s+2)}$$

It is desired that dominant closed-loop poles provide damping ratio $\zeta = 0.5$ and have an undamped natural frequency $\omega_n = 4$ rad/sec. Velocity error constant K_v is required to be greater than 4.
(a) Verify that only gain adjustment can't meet these objectives.
(b) Design a lead compensator to meet the objectives.
(c) Using GUI, determine the peak overshoot and settling time of the lead-compensated system
(Ans: $M_p = 21\%$; $t_s = 0.02$ sec)
(d) Design a lag compensator to meet the objectives (Refer Example 7.10).

A.11 *Example 10.3 revisited.*
A unity-feedback system has open-loop transfer function

$$G(s) = \frac{K}{s(s+1)}$$

It is desired to have the velocity error constant $K_v = 10$. Furthermore, we desire that the phase margin of the system be atleast 45°. Design a phase-lag compensator to achieve these specifications.
Are your results different from the ones given in the text? Why?
Using GUI, determine the peak overshoot and settling time of the lag-compensated system.
(Ans: $M_p = 28\%$; $t_s = 14$ sec)

A.12 *Example 10.1 revisited.*
Repeat Problem A.11 under the constraint that we use phase-lead compensation to achieve the performance specifications. (Ans: $M_p = 29\%$; $t_s = 1.77$ sec)

A.13 *Example 8.16 revisited.*
A unity-feedback system has open-loop transfer function

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

Determine gain crossover frequency, phase crossover frequency, phase margin and gain margin when (i) $\tau_D = 0$ (ii) $\tau_D = 1$.

A.14 *Example 11.2 revisited.*
For a unity-feedback system with plant transfer function $G(s) = 1/s^2$, show that the cascade compensator $D(s) = 0.81\,(s + 0.2)/(s + 2)$ meets the specifications: $\zeta = 0.7$, $\omega_n = 0.3$. Plot step response of the compensated system. Now discretize the compensator (sampling time $T = 1$ sec) and plot step response of the discretized system with digital compensator. Comment upon the discrepancy with respect to performance achieved using analog design.

A.15 *Example 12.9 revisited*
Show that the Inverted Pendulum described in this example is unstable. Also show that state-feedback

$$u(t) = -\mathbf{k}\,\mathbf{x}(t)$$

$$\mathbf{k} = [-15.4785 \quad -2.9547 \quad -0.0705 \quad -0.2820]$$

stabilizes this system.
For initial conditions

$$\mathbf{x}(0) = [0.1 \quad 0 \quad 0 \quad 0]^T$$

and zero external input, simulate the feedback system.

A.16 *Review Example 10.4 revisited*
In the electromechanical servo system of Fig. 10.34, the compensator

$$D(s) = 17.05(s + 5)/(s + 0.2)$$

(a) Simulate the system for step commands, ramp commands, and step disturbances.
(b) Set the parameters of "Command" and "Disturbance" blocks to 0. Insert a "Step" source at the error point e (refer Fig. 10.34). Connect "To Workspace" block to the output of "Armature" block and study the variation of armature current when error step inputs are given. Determine the range of error for which the armature current does not exceed 5A. For this range of error, determine the range of the amplifier output voltage e_a (Fig. 10.34).

A.17 *Review Example 10.4 revisited*
In the electromechanical servo system of Fig. 10.34, the compensator

$$D(s) = 17.05(s + 5)/(s + 0.2)$$

(a) Simulate the system for step commands of value higher than 0.29 rad.
(b) Insert the saturation nonlinearity block at the output of the "Amp gain" block. Set the parameters of the saturation nonlinearity as per Fig. 10.36. Simulate this system for step comands of values greater than 0.29 rad.
(c) Compare the responses in (a) and (b) for the same value of step command.