Appendix

A

MATLAB Aided Control System Design: Conventional Control

This appendix also provides guidance to the reader to integrate the learning of control system analysis and design material with the learning of how one computes the answers with MATLAB software package. The appendix is not meant to be a substitute for the MATLAB software manuals. Mathworks [152] provides extensive documentation in both printed and online format to help you learn about, and use, all the features of MATLAB. The online **help** provides task-oriented, and function reference information. Our attempt here is only to expedite the process of making full use of the power of the package for control system design problems.

The MATLAB statements/responses given in this appendix are from MATLAB version 7 and Control System Toolbox version 6.1, on the Windows platform. Refer [151] for an introduction to the MATLAB environment.

A sequence of characteristic steps of MATLAB-aided control system design follows.

Make a System Model

For time-invariant systems, mathematical model-building based on physical laws normally results in a set of differential equations. These equations, when rearranged as a set of first-order differential equations, result in a state-space model of the following form:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{u}) \\ \mathbf{y} &= \mathbf{h}(\mathbf{x}, \mathbf{u}) \end{aligned} \tag{A.1}$$

where $\mathbf{f}(\cdot)$ and $\mathbf{h}(\cdot)$ are nonlinear functions of their arguments; $\mathbf{x}(t)$ is the $n \times 1$ internal state vector, $\mathbf{u}(t)$ is the $p \times 1$ control input vector, and $\mathbf{y}(t)$ is the $q \times 1$ measured output vector. Overdot represents differentiation with respect to time t. The first equation, called the state equation, captures the dynamical portion of the system and has memory inherent in the n integrators. The second equation, called the output or measurement equation, represents how we chose to measure the system variables; it depends on the type and availability of sensors.

Model-building and its validation will require *simulation* of the model. The nonlinear state-space equations of the form (A.1) are very easy to simulate on a digital computer. MATLAB provides many Runge–Kutta numerical integration routines for solving ordinary differential equations; the function **ode23** usually sufffices for our applications. Use of **ode23** function for solving ordinary differential equations of the type (A.1) will be demonstrated later.

Linearization of equations of the form (A.1) leads to a model of the form given below:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \\ \mathbf{y} &= \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u} \end{aligned} \tag{A.2}$$

The $n \times 1$ vector **x** is the state of the system, **A** is the constant $n \times n$ system matrix, **B** is the constant $n \times p$ input matrix, **C** is the constant $q \times n$ output matrix, and **D** is the constant $q \times p$ matrix.

While considering only single-input, single-output (SISO) problems, we take the number of inputs, p, and the number of outputs, q, to be one. For SISO problems, the state variable model may be expressed as

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{b}u \\ y &= \mathbf{c}\mathbf{x} + du \end{aligned} \tag{A.3}$$

Note that matrices **B**, **C**, and **D** of the MIMO (multi-input, multi-output) representation, now become vectors **b** and **c**, and scalar d, respectively; and the input vector **u** and output vector **y** of the MIMO representation, now become scalar variables u and y respectively.

Validation of the linear model with experimental data or simulation data obtained from (A.1), will require simulation of Eqns (A.2)/(A.3). MATLAB provides many functions for simulation of linear models. We consider here some functions important from the control-engineering perspective.

For the state-space representation (A.2)/(A.3), the data for the model consists of four matrices. For convenience, the MATLAB provides customized data structure (LTI object). This is called the **SS** object. This object encapsulates the model data and enables you to manipulate the LTI system as a single entity, rather than as a collection of data vectors and matrices.

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

```
sys = ss(A,B,C,D)
```
(A.4)

creates the state-space model (A.2).

When the four matrices of the model (A.3) are entered,

```
sys = ss(A,b,c,d)
```
(A.5)

creates a state-space model for SISO systems.

Depending on the type of design model you use, the data for your model may consist of a simple numerator/denominator pair for transfer functions or four matrices for state-space models. MATLAB provides LTI objects **TF** for transfer functions.

An LTI object of the type **TF** is created whenever you invoke the construction function **tf**.

```
sys = tf(num,den)
```
(A.6)

num and **den** vectors specify $n(s)$ and $d(s)$ respectively, of the transfer function $G(s) = n(s)/d(s)$.

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 $G(s)$ 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 deadtime (input-output delay). **TF** object for transfer functions with deadtime can be created using the syntax

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

Discretize the Design Model

There are various standard methods for discretization of continuous-time models. None of these methods is exact for all types of inputs, because no sampled system has access to the input time history between samples. In essence, each approximation makes a different assumption about what the continuous input is doing between samples. MATLAB software has functions that allow use of various approximations. We consider here a couple of functions important from the control-engineering perspective.

State-space model of a discrete-time SISO system is of the form

$$\mathbf{x}(k+1) = \mathbf{F}\mathbf{x}(k) + \mathbf{g}u(k)$$
$$y(k) = \mathbf{c}\mathbf{x}(k) + du(k) \tag{A.7}$$

Construction of the **SS** object for this discrete-time model, requires four matrices **F, g, c** and d, and the sampling interval T.

```
sysd = ss(F,g,c,d,'T')
```

Transfer function model of a discrete-time SISO system is of the form

$$G(z) = \frac{n(z)}{d(z)} = \frac{num}{den}$$

Construction of the **TF** object for this discrete-time model requires *num* and *den* polynomials in z, and the sampling interval T.

```
sysd = tf(num, den,'T')
```

For a continuous-time model **sysc**, the command

```
sysd = c2d(sysc,T) % T is sampling period in seconds
```

performs ZOH conversion by default.

sysc is continuous-time state-space system (A.3) and **sysd** is the discrete-time system (A.7), assuming a zero-order hold on the input—the control input is assumed piecewise constant over the sample time T.

```
[F,g,c,d] = ssdata(sysd)
```

c2d can also be used with transfer function models. Continuous-time system **sysc** representing

$$G(s) = \frac{num}{den}$$

gets converted to the discrete-time system **sysd**, representing

$$G_{h0}G(z) = \frac{num\,z}{den\,z}$$

```
[numd,dend] = tfdata(sysd,'v')
```

returns numerator and denominator of transfer function as row vectors.
To use alternative conversion schemes, specify the method.

```
sysd = c2d(sysc,T,'tustin') % Use Tustin approximation
```

The function **c2d(sysc,T,'tustin')** converts a continuous-time system to discrete-time system using trapezoidal rule for integration, also called the bilinear transformation. (The function **d2c** is an inverse operation—it converts discrete-time models to continuous-time form).

Try Lag-Lead Design/State-Space Design

The function **bode(sys)** generates the Bode frequency-response plot for LTI model **sys**. 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 utilizing the **logspace** function. When invoked with left-hand arguments,

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

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

The function **margin** determines gain margin, phase margin, gain crossover frequency and phase crossover frequency.

The function **bode** handles both continuous-time and discrete-time models. For continuous-time models, it computes the frequency response by evaluating the transfer function $G(s)$ on the imaginary axis $s = j\omega$. For discrete-time models, the frequency response is obtained by evaluating the transfer function $G_{h0}G(z)$ on the unit circle $z = e^{j\omega T}$; where T is the sample time.

$$\text{mag}(\omega) = |G_{h0}G(e^{j\omega T})|$$

$$\text{phase}(\omega) = \angle G_{h0}G(e^{j\omega T})$$

The discrete-time model of the plant $G_{h0}G(z) = num/den$, may be transformed with bilinear mapping using the MATLAB function **d2c**. Construction of discrete-time model requires numerator polynomial *num* in z, denominator polynomial *den* in z, and value of the sampling interval T.

sysd = tf(num,den,'T')

Conversion to w-domain is given by

sysw = d2c(sysd,'tustin')

The Nichols frequency-response plot can be generated using **Nichols** function; the frequency range is user-selectable. A Nichols chart grid is drawn on the existing plot with the **ngrid** function. These functions are applicable to both continuous-time and discrete-time systems.

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.

MATLAB function **rlocus(sys)** calculates and plots the root locus of the open-loop SISO model **sys**. If **sys** has 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.

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 crosshair 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 functions **rlocus** and **rlocfind** work with both the continuous-time and discrete-time SISO systems. The functions **sgrid/spchart** and **zgrid/zpchart**, are used for ω_n and ζ grid on continuous-time and discrete-time root locus, respectively.

MATLAB provides many functions for simulation of transfer functions. For the transfer function model **sys = tf(num,den), step(sys)** will generate a plot of unit-step response $y(t)$ from the transfer function

$$\frac{Y(s)}{U(s)} = G(s) = \frac{num}{den}$$

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 = 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.
```

MATLAB provides similar functions for simulation of discrete-time systems. **step(sys)** will generate a plot of unit-step response $y(k)$ of the discrete-time system (transfer function model) **sys**. The number of sample points is automatically determined when time is not explicitly included in the command.

If you wish to supply the sample points vector at which the response will be computed, the following command is used.

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

You can specify **t** as a vector of sample points:

```
t = 0:T:tfinal
```

where **T** is the sample time.
When invoked with left-hand arguments, no plot is generated on the screen; it is necessary to use a **plot** command to see the response curves.

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.

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 matrix **c**; the output is observability matrix **V**. The function **rank** gives the controllability and observability properties.

Pole-placement design may be carried out using the MATLAB function **acker**. However, the computation of the controllability matrix has very poor numerical accuracy and this carries over to Ackermann's formula. The function **acker** can be used for the design of SISO systems with a small (≤ 5) number of state variables. For more complex cases, a more reliable formula is available, implemented in MATLAB with the function **place**. A modest limitation on **place** is that none of the desired closed-loop poles may be repeated, that is, the poles must be distinct; a requirement that does not apply to **acker**.

The MATLAB function **lyap** solves Lyapunov matrix equation. The solution to discrete matrix Lyapunov equation is found with **dlyap**. The function **lqr** solves the linear quadratic regulator problem and associated Riccati equation. Discrete linear quadratic regulator design is carried out using the function **dlqr**.

MATLAB provides many functions for simulation of state-space models. For model **sys** in (A.5), **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 in vector u
lsim(sys,u,t,x0)  % having length (t) rows
```

For MIMO models (A.4), these functions produce an array of plots.

MATLAB provides similar functions for simulation of discrete-time state-space models. **step(sys)** will generate a plot of unit-step response $y(k)$ of the discrete-time system state-space model **sys**. Zero initial state is assumed. The number of sample points is automatically determined when time is not explicitly included in the command.

If you wish to supply the sample-points vector at which the response will be computed, the following command is used.

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

You can specify **t** as a vector of sample points:

```
t = 0 : T : tfinal
```

where **T** is the sample time.

You can analyze the time response using the **GUI** (graphical user interface) for viewing and manipulating response plots of LTI models.

After reaching the best compromise on controller design choice, the next step is to build a computer model, and compute (simulate) the performance of the design. At this stage, we are required to compute a digital equivalent of the analog controller. This allows the final design to be implemented using digital processor logic.

Given the analog controller

$$D(s) = num/den$$

the following commands give the discrete equivalent

$$D(z) = numz/denz$$

of the controller, with T as the sampling period.

```
sysc = tf(num,den)
sysd = c2d(sysc,T,'tustin')
```

Simulate the Performance of the Design

After reaching the best compromise among process modification, actuator and sensor selection, and controller design choice, run a computer model of the system. This model should include important nonlinearities—such as actuator saturation, and the 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.

To use the MATLAB numerical integration routines, the system dynamics must be written into an **M-file**. The state-space description makes this very direct; in fact, ordinary differential equation solver function **ode23** , requires the dynamics in state-space form (A.1).

As an example, consider Van der Pol oscillator which has dynamics

$$\ddot{y} + \alpha(y^2 - 1)\,\dot{y} + y = u$$

Defining the states as x_1 = position, x_2 = velocity, we get

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = \alpha(1 - x_1^2)\,x_2 - x_1 + u$$

For the Van der Pol oscillator, the required **M-file** is

```
function xdot = vdpol (t, x)
alpha = 0.8; u = 0;
xdot = [x(2); alpha*(1 – x(1)^2)*x(2) – x(1) + u];
```

where it is assumed that $u(t) = 0$. Now the sequence of commands required to invoke **ode23** and obtain time history plots, for instance, over a time horizon of 50 sec is

```
t0 = 0; tf = 50;
x0 = [0.1;0.1];
[t,x] = ode23('vdpol', [t0 tf], x0);
plot(t, x)
```

The phase-plane plot of x_1 versus x_2 is obtained using the command:

```
plot(x(:,1), x(:,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 computational 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. For a technical introduction to Simulink, read the MATLAB document "Using Simulink" [152]. To help you start using Simulink quickly, we describe here the simulation process through 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 labelled **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 labelled **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). This block is useful for phase-plane analysis. 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. We will use this block for phase-plane analysis. 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, 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 then zero). We can use this block to represent on-off nonlinearity.

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" [152].

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 that opens up, 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 phase-plane analysis of nonlinear system of Fig. A.1.

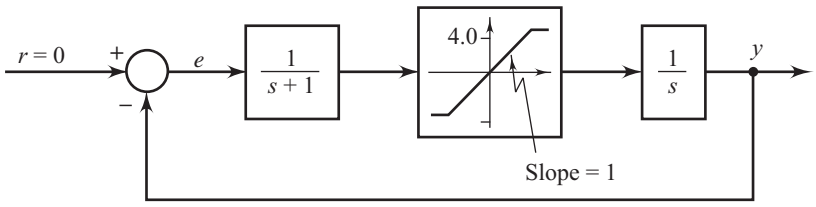

Fig. A.1

With the **Discontinuities** subnode of Simulink node expanded, move the pointer and click the block labelled **Saturation**, and while keeping the mouse button pressed down, drag the block and place it inside the Simulation Window, and release the mouse button.

With the **Control System Toolbox** node expanded, click the block labelled **LTI system**, drag to the Simulation Window and place it on one side of the Saturation block. Duplicate **LTI system** on the other side of Saturation block.

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 the block labeled **Sum** from the **Math Operations** subnode of Simulink node, the block **Constant** from the **Sources** subnode of Simulink node, the blocks **XY Graph** and **To Workspace** from **Sinks** subnode of Simulink node, and the block **Derivative** from the **Continuous** subnode of Simulink node.

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 the mouse button.

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.

This gives us a generic diagram, because we have not yet specified the LTI system, nor set parameter values of saturation, reference input, and error detector. Our next priority is to go into each of these blocks and set the parameters that correspond to our specific nonlinear system. In addition, we need to set some simulation parameters.

We begin with the reference input to the feedback system by double-clicking on the block labeled **Constant** in the Simulation Window. A dialog box pops up. Only one parameter need to be set: **constant value**. Set the value to **0** since our reference input is zero. When we are done, we click **OK**.

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.

Next, we set the LTI system blocks. There are two blocks on the two sides of the saturation nonlinearity. The first block has the transfer function $1/(s + 1)$, and the second block has the transfer function $1/s$. To carry out simulation study with respect to initial condition –1.6 on output, we convert the transfer functions to state-space form. Enter the initial condition **–1.6** in the dialog box of the second block.

Saturation block dialog box requires upper limit and lower limit of saturation. **0.4** and **–0.4** are the values as per our problem.

Next, we need to set the parameters for the **XY Graph** block. Dialog box requires ***x*-min**, ***x*-max**, ***y*-min** and ***y*-max**. The values [–1 2 –2 1] may be entered.

To Workspace block requires variable name and the format. We use **Array** format for our data and enter variable names **x_1** and **x_2** in the dialog boxes.

Finally, we need to set the parameters for the simulation run. We move the pointer to the menu labelled **Simulation**, and enter configuration parameters: **start time**, **stop time**, in the dialog box.

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.

At this point in the simulation process, we have generated the appropriate Simulink block diagram (shown in Fig. A.2) 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**. A new window that shows the phase trajectory pops up.

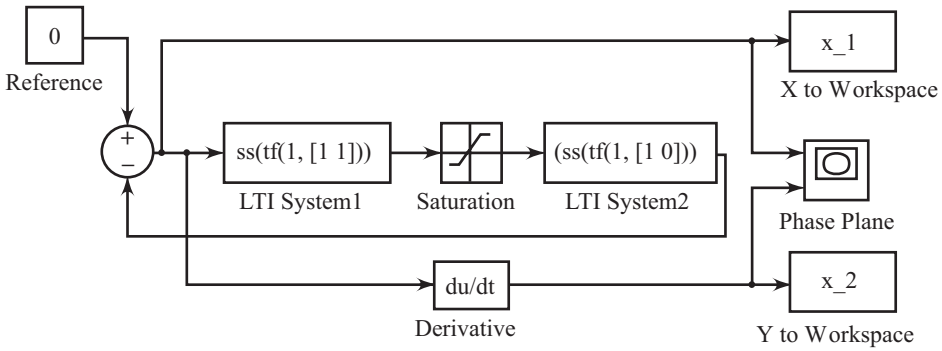

Fig. A.2

You may now execute the following program in MATLAB workspace.

```
figure (1);
plot (x1, x2); grid;
axis ([–1  2  –2  1]);
hold on
```

Resimulate for an initial condition of –0.74 and plot the phase trajectory.

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 **SimulinkFigA.2** from URL: **http:/www.mhhe.com/gopal/dc3e**. Open this file 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 are given at the URL:

URL: http:/www.mhhe.com/gopal/dc3e

as help to these problems. Open these files in MATLAB environment. In attempting a problem, the reader can use the MATLAB commands given 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 simlink 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 Example 3.1 revisited Consider a unity-feedback system with open-loop transfer function

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

(a) Plot the step response of the feedback system and determine error constants K_p, K_v and K_a.
(b) Discretize the system (sampling interval $T = 1$ sec) and plot the step response of the resulting feedback system. Also determine the error constants.
(c) Approximate the sampled system by an equivalent analog system with input delay of $T/2$. Plot the step response of the resulting analog feedback system.
(d) Using **GUI**, determine peak overshoot and settling time of analog and sampled systems.

A.2 Example 3.2 revisited Consider a unity-feedback system with open-loop transfer function

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

(a) Sketch root locus plot and determine the range of gain K, for which the system is stable.
(b) Discretize the system (sampling interval $T = 0.4$ sec) and sketch root locus plot. Find the range of gain K for which the system is stable.
(c) Repeat (b) for $T = 3$ sec.

A.3 Example 3.3 revisited Consider a system with transfer function

$$G(s) = \frac{e^{-1.5\,s}}{s\,(s+1)}$$

Discretize this system (sampling time $T = 1$ sec) and report the result in zero-pole-gain form.

A.4 Example 2.17 revisited A unity-feedback sampled-data system (sampling interval $T = 0.04$ sec) has plant transfer function

$$G(s) = \frac{10}{(1+0.5s)(1+0.1s)(1+0.05s)}$$

An approximating analog system is a unity-feedback system with plant transfer function $G(s)e^{-Ts/2}$. Show that the analog controller

$$D(s) = \frac{0.67s + 1}{(2s + 1)}$$

meets the specification: phase margin ≥ 40°. Determine the bandwidth of the compensated system.

Discretize the design, and analyze the step response of the digital system using GUI (*Ans*: Peak overshoot 13%; Settling time 1.16 sec)

A.5 Example 4.3 revisited A unity-feedback system has open-loop transfer function

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

It is desired to have the velocity error constant $K_v = 10$. Furthermore, we desire that the phase margin of the system be about 40° and bandwidth about 5.5 rad/sec. Design a digital control scheme ($T = 0.1$ sec) to meet these specifications.

Using **GUI**, determine peak overshoot and settling time from the step response of the feedback system (*Ans*: Peak overshoot 34%; Settling time 3 sec)

Are your results different from the ones given in the text? Why?

A.6 Example 4.4 revisited Repeat Problem A.5 under the constraint that we use phase-lead compensation to achieve the following performance specifications:

(i) $K_v = 10$
(ii) Phase margin = 40°
(iii) Bandwidth = 12 rad/sec (*Ans*: Peak overshoot 35%; Settling time 1 sec)

A.7 Example 4.7 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 undamped natural frequency $\omega_n = 4$ rad/sec. Velocity error constant K_v is required to be about 2.5.

Design a digital control scheme ($T = 0.2$ sec) for the system to meet these specifications.

Perform simulation study on the compensated system using **GUI**

(*Ans*: Peak overshoot 15%; Settling time 2.2 sec)

A.8 Example 4.8 revisited The plant of sampled-data system of Fig. 4.30 is described by the transfer function

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

The sampling period is 1 sec.

The problem is to design a digital controller $D(z)$ to realize the following specifications: overshoot < 16%; settling time < 10 sec; $K_v \geq 1$.

Use frequency responce plots of $G_{h0}G(e^{j\omega T})$ for the design.

A.9 In the following, we point the reader to important matrix functions in MATLAB. Access the description of these functions from the **help file**, and execute each function, taking suitable data from the text.

Identity matrix	:	**eye(n)**
Dimensions	:	**size(A)**
Utility matrices	:	**ones(n), ones(m,n), ones(size(A)), zeros(n), zeros(m,n), zeros(size(A))**

Complex-conjugate transpose	:	**ctranspose(A); A′**
Non-conjugate transpose	:	**transpose(A); A′**
Determinant	:	**det(A)**
Inverse	:	**inv(A)**
Rank	:	**rank(A)**
Trace	:	**trace(A)**
Spectral norm (Largest singular value)	:	**norm(A)**
Euclidean norm of a vector	:	**norm(x)**
Condition number with respect to inversion	:	**cond(A)**
Eigenvalues	:	**eig(A)**
Eigenvectors	:	**[P,A1] = eig(A)**
Characteristic equation	:	**poly(A)**
Matrix exponential	:	**expm(A)**

A.10 Given the transfer function

$$G(s) = \frac{s}{s^3 + 2s^2 + 2.5s + 0.5}$$

(a) Obtain a state-space model **sys**, equivalent to the given $G(s)$.
(b) Discretize the model **sys** (sampling interval $T = 0.1$ sec) to obtain **sysd**.
(c) Simulate and plot the response of the models **sys** and **sysd**, when the input is

$$u(t) = \begin{cases} 2 \; ; \; 0 \le t \le 2 \\ 0.5; \; t \ge 2 \end{cases}$$

and the initial condition is $\mathbf{x}(0) = [1 \quad 0 \quad 2]^T$.

A.11 Example 7.2 revisited Linearized equations governing the inverted pendulum system of Fig. 5.16 are

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{b}u$$
$$\mathbf{x} = [\theta \quad \dot{\theta} \quad z \quad \dot{z}\,]^T$$

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 16.3106 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -1.0637 & 0 & 0 & 0 \end{bmatrix}; \; \mathbf{b} = \begin{bmatrix} 0 \\ -1.4458 \\ 0 \\ 0.9639 \end{bmatrix}$$

(a) Show that the open-loop system is unstable.
(b) Design state feedback $u = -\mathbf{k}\mathbf{x}$ that results in closed-loop poles at $-0.7999 \pm j\,1.5618, -10, -11$. Note that the complex-conjugate poles correspond to the requirement of peak overshoot of 20% and settling time of 5 sec (2% tolerance).
(c) Simulate the feedback system; given initial state

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

Check the robustness of your design through simulation study using nonlinear plant model given by Eqns (5.104).

(d) For this system, design an observer with the onserver poles placed at $-2, -2 \pm j\,1, -3$.
(e) Simulate the observer-based feedback system for the initial conditions given in part (c). Check the robstness of your design through simulation study using nonlinear plant model given by Eqns (5.104).

A.12 Example 7.1 revisited The plant model of a satellite attitude control system (Refer Figs 7.3–7.4) is

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{b}u$$
$$y = \mathbf{c}\mathbf{x}$$

with
$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}; \mathbf{b} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}; \mathbf{c} = [1 \quad 0]$$

(a) Design state feedback $u = -\mathbf{k}\mathbf{x}$ that results in closed-loop poles at $-4 \pm j4$.
(b) Assuming that the state vector $\mathbf{x}(t)$ is measurable, simulate the feedback system for $\mathbf{x}(0) = [1 \quad 0]^T$.
(c) Consider, now that state measurements are not practical. Design a state observer that yields estimated states $\tilde{\mathbf{x}}(t)$. Place the observer poles at $-10, -10$.
(d) Obtain state variable model of the compensator by cascading the state feedback control law and the state observer. Find the transfer function of the compensator.
(e) Set up state model of the form (7.57) for the observer-based regulator system, and simulate the model (Note that $\mathbf{x}(0) = [1 \quad 0]^T$ leads to $\tilde{\mathbf{x}}(0) = [1 \quad 0]^T$ when $\hat{\mathbf{x}}(0) = \mathbf{0}$).

A.13 Example 7.13 revisited The plant model of a satellite attitude control system (Refer Figs. 7.3–7.4) is

$$\mathbf{x}(k+1) = \mathbf{F}\mathbf{x}(k) + \mathbf{g}u(k)$$
$$y(k) = \mathbf{c}\mathbf{x}(k)$$

with
$$\mathbf{F} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix}; \mathbf{g} = \begin{bmatrix} T^2/2 \\ T \end{bmatrix}; T = 0.1 \text{ sec}$$
$$\mathbf{c} = [1 \quad 0]$$

The reference input θ_r is a step function.
Design state feedback $u = -k_1(x_1(k) - \theta_r) - k_2 x_2(k)$ that results in deadbeat response.
Simulate the feedback system for a unit-step input θ_r.

A.14 Reconsider the inverted pendulum regulator problem raised in Problem A.11, wherein you designed state feedback control based on pole-placement.

Now design a state-feedback control law that minimizes the performace index

$$J = \frac{1}{2}\int_0^\infty (\mathbf{x}^T\mathbf{Q}\mathbf{x} + u^T R u)dt$$

with
$$\mathbf{Q} = \begin{bmatrix} 150 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 40 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}; R = 1$$

Simulate the feedback system for initial state $\mathbf{x}(0) = [0.1\ 0\ 0\ 0]^T$.

A.15 Reconsider the inverted pendulum system of Problem A.11.

(a) Discretize the plant model (sampling interval $T = 0.1$ sec).
(b) Introduce integral state in the plant equations (Refer Eqn. (7.109)) and show that the augmented system is controllable.
(c) Design state feedback with integral control that minimizes the performance index

$$J = \frac{1}{2}\sum_{k=0}^{\infty}[\mathbf{x}^T(k)\mathbf{Q}\mathbf{x}(k) + u^T(k)Ru(k)]$$

with

$$\mathbf{Q} = \begin{bmatrix} 10 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 100 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}; \; R = 1$$

(d) Simulate the digital servo (Refer Fig. 7.17) for a step input.

A.16 Review Example 9.2 revisited Figure 9.49a shows the block diagram of a nonlinear system with saturation nonlinearity.

(a) Sketch the Nyquist plot for the linear transfer function

$$G(s) = \frac{1}{s(1 + 2s)(1 + s)}$$

(b) Superimpose on this plot, the plot of describing function of saturation nonlinearity.
(c) Show the existence of a stable limit cycle and determine its amplitude and frequency.
(d) Drag the following blocks from Simulink block libraries:
  (i) **Sum** from "Math Operations" subnode of Simulink node;
  (ii) **Saturation** from "Discontinuities" subnode of Simulink node;
  (iii) **LTI system** from "**Control System Toolbox** node": and
  (iv) **Scope** from "**Sinks**" subnode of Simulink node.

Setup a simulation block diagram as per the feedback structure given in Fig. 9.49a. Simulate the system for an initial condition of $\mathbf{x}(0) = [5 \quad 0 \quad 0]^T$. The Simulink response shows a limit cycle. Determine the amplitude and frequency of the limit cycle and compare these parameters with the ones obtained in part (c).

A.17 Problem A.11 revisited In Problem A.11, you have designed a controller and an observer using linearized model of the pendulum. Apply the controller and the observer to the nonlinear plant model given by Eqns 5.104. Use MATLAB's Simulink for this simulation study.

A.18 Section 10.2 revisited This section details a case study on feedback linearization of a 2-link robot manipulator. Carry out MATLAB-aided design. Simulate the feedback system and compare your result with that given in Fig. 10.3.

A.19 Section 10.3 revisited Carry out MATLAB-aided design of model-reference adaptive control system (Fig. 10.5) for the given plant. simulate the system and compare your result with that given in Fig 10.6.

A.20 Section 10.4 revisited Carry out MATLAB-aided design of self-tuning regulator (Fig 10.7) for the given plant. Simulate the system and compare your result with that given in Fig 10.8.

A.21 Section 10.5 revisited This section details a case study on sliding-mode control of a 2-link robot manipulator. Carry out MATLAB-aided design. Simulate the feedback system and conpare your result with that given in Fig 10.10.

A.22 Problem A.18 revisited In Problem A.18, you have designed a PD controller for a 2-link robot manipulator after feedback-linearization of the plant model. Apply the feedback-linearization loop and PD-controller to the nonlinear plant model. Use MATLAB's Simulink for this simulation study.