Appendix A

Introduction to MATLAB, SIMULINK and the Control Systems Toolbox

MATLAB, a registered trademark of MathWorks, Inc. [1], is a high-level programming language which uses *matrices* as the basic numerical entities (rather than *scalars*, as in the low-level programming languages such as BASIC, FORTRAN, PASCAL, and C). In other words, MATLAB allows us to *directly* manipulate matrices – such as adding, multiplying, inverting matrices, and solving for eigenvalues and eigenvectors of matrices (see Appendix B). If similar tasks were to be performed by a low-level programming language, many programming statements constituting scalar operations would be required for even the simplest matrix operations. Hence, MATLAB is ideally suited for linear algebraic computations involving matrices, such as multivariable control design and analysis. Furthermore, MATLAB contains a library of many useful functions – both basic functions (such as trigonometric, hyperbolic, and exponential functions), and specialized mathematical functions – along with an advanced facility for plotting and displaying the results of computations in various graphical forms. In addition, MATLAB is supplemented by various special application *toolboxes*, which contain additional functions and programs. One such toolbox is the *Control System Toolbox*, which has been used throughout this book for solving numerical examples for the design and analysis of modern control systems. The *Control System Toolbox* is available at a small extra cost when you purchase MATLAB, and is likely to be installed at all the computer centres that have MATLAB. If your university (or organization) has a computer center, you can check with them to find out whether the *Control System Toolbox* has been installed with the MATLAB. If you are an engineering/science student, or a practicing engineer, with interest in solving control problems, it is worth having access to both MATLAB and its *Control System Toolbox*, which are available in *student editions* for most platforms supporting WINDOWS or UNIX. In this book, it is assumed that you have the *Control System Toolbox* installed in your MATLAB directory, and we draw upon the special functions and programs contained in the *Control System Toolbox* for solving the numerical examples and exercises. You can devise your own computer programs in MATLAB (called *M-files*) for solving special problems. Some new M-files have been provided in the book for solving a range of control problems.

A.1 Beginning with MATLAB

Here we will discuss how to familiarize ourselves with MATLAB. For further details, you are referred to the MATLAB *User's Guide* [1]. The User's Guide contains necessary information about system requirements, installing and optimizing MATLAB. Once you have MATLAB installed and running on your computer, a *command line* appears on the screen with the prompt (>>) after which you can issue MATLAB commands. After a command is issued at the prompt, you have to press the <enter> key for the command to be executed. All the commands executed at the command line, and the variables computed in those commands, are stored automatically and can be recalled, unless you end the MATLAB session. A MATLAB command consists of one or several MATLAB *statements*. Each MATLAB statement could be of one of the following forms:

```
>> variable = expression

or

>> expression
```

A *variable* is usually a matrix to be computed, while the *expression* is the mathematical operation by which the variable is to be computed. A *variable* can have a *name* beginning with a letter, followed by up to 18 letters, digits, or underscores. The names in MATLAB are *case sensitive* (i.e. upper and lower case are distinguished). If we omit the '*variable* =' from a statement, MATLAB automatically creates a variable named *ans*, which is abbreviation for *answer*. For example, consider the following matrix, **A**, of size (4 × 3) and another matrix, **B**, of size (3 × 2) which have to be multiplied:

$$\mathbf{A} = \begin{bmatrix} -1 & -4 & 0 \\ 18 & 26 & 7 \\ 9 & 6 & -3 \\ 11 & 0 & 4 \end{bmatrix}; \quad \mathbf{B} = \begin{bmatrix} 20 & 8 \\ 2 & 0 \\ 5 & 13 \end{bmatrix} \tag{A.1}$$

We must assign values to the two matrices at the command line by two separate *statements*. Each statement uses *square brackets* to denote the beginning and end of the matrix, and separates two consecutive elements in each row by a *space*. Two consecutive rows are separated by a *semi-colon*. The entire command assigning values to the matrices **A** and **B** is thus issued as follows:

```
>> A=[-1 -4 0; 18 26 7; 9 6 -3; 11 0 4], B=[20 8; 2 0; 5 13] <enter>
```

Note that the two statements in the above command line have been separated by a *comma*. This command produces the following result on the screen:

```
A =
   -1   -4   0
   18   26   7
   9    6    -3
   11   0    4
```

```
B =
  20   8
   2   0
   5   13
```

which confirms that the matrices have been correctly entered. If you do not wish to see the results of your command on the screen, you must end *each statement* in the command line by a semi-colon. For example, the following command will store the matrices **A** and **B** in the memory, but would *not* result in their screen print-out:

```
>> A=[-1 -4 0; 18 26 7; 9 6 -3; 11 0 4]; B=[20 8; 2 0; 5 13]; <enter>
```

The multiplication of the matrices **A** and **B** (already entered into the memory of the MATLAB work-space) can be carried out using the symbol *, and the product, **AB**, can be stored as a *third* matrix, **C**, as follows:

```
>> C = A*B <enter>

C =
 -28   -8
 447  235
 177   33
 240  140
```

In this manner, you can carry out all other basic matrix operations, such as addition, subtraction, transposition, inversion, left-division, right-division, raising a matrix to a power, transcendental and elementary matrix functions, which are briefly described in the following section. Each of the *elements* of a matrix could be a MATLAB *expression*, such as

```
>>x = [1/4 25+sqrt(8.7); 0.5*sin(1.3) 7*log(0.68)] <enter>

x =
0.2500   27.9496
0.4818   -2.6996
```

where / denotes division, + denotes addition, `sqrt(.)` denotes the positive square-root, `sin(.)` denotes the sine function, and `log(.)` denotes the natural logarithm. An element of a matrix can be referenced with indices inside parentheses, (i, j), indicating ith row and jth column position, such as

```
>>x(2,2) <enter>

ans =
 -2.6996
```

is the (2, 2) element of the matrix, **x**. If we assign a value to an element of a matrix with indices *larger* than the size of the previously stored value of the same matrix, the size of

the matrix is automatically *increased* to the new dimension, and all undefined elements are set to zero. For example,

```
>>x(3,3)=10 <enter>
```

results in the following value of **x**:

```
x =
  0.2500   27.9496   0
  0.4818   -2.6996   0
     0      0        10.0000
```

We can extract smaller matrices from the rows and columns of a larger matrix by using the *colon*. For example, a matrix **y** defined as the matrix formed by taking elements contained in the *first two rows* and the *second and third columns* of **x** is formed as follows:

```
>>y = x(1:2, 2:3) <enter>

y =
  27.9496   0
  -2.6996   0
```

The *colon* can also be used to generate elements with equal spacing, such as

```
>> x = 0:5/4:5 <enter>
```

which results in the following row-vector with elements from 0 to 5 with increments of 1.25:

```
x =
     0 1.2500 2.5000 3.7500 5.0000
```

MATLAB accepts numbers in various *formats*, such as the conventional decimal notation, a power-of-ten scale factor, or a complex unit as a suffix. For example, the following assignment of a matrix is acceptable in MATLAB:

```
>>A = [1 -100i 0.0003; 9.87e5 1.5-4.69j 7.213e-21; 3+5e-4i -7.019e-3 2j] <enter>
```

where i (or j) denotes the imaginary part of a complex number (i.e. square root of -1), and e followed by up to three digits denotes the power of 10 to which a number is raised. The accuracy of floating-point arithmetic in MATLAB is about 16 significant digits, with a range between 10^{-308} and 10^{308}. Any number falling outside this range of floating-point arithmetic is called *NaN*, which stands for *not a number*. You can select from various formats available in MATLAB for printing your results, such as *short* (fixed-point format with 5 digits), *long* (fixed-point format with 15 digits), *short e* (floating-point format with 5 digits), *long e* (floating-point format with 15 digits), *hex* (hexadecimal), and *rat* (numbers approximated by ratios of small integers). The variables *ans* (answer) and *eps*

(2^{-52}) are treated as *permanent variables* in MATLAB, and cannot be cleared or re-assigned in the memory. Some built-in functions return commonly used variables, such as *pi*, and *inf* which stand for π, and ∞, and should not be re-assigned other values in a computation. For ease of programming, MATLAB also provides the function matrices called *ones*, *zeros*, and *eye*, which stand for a matrix of *all* elements equal to 1, a matrix with *all zero* elements, and an *identity* matrix, respectively. The sizes of these matrices can be specified by the user, such as *ones*(3,4), *zeros*(2,6), or *eye*(5).

MATLAB supports help on all its commands, and you can receive help by typing

```
>> help  command <enter>
```

or merely type

```
>>help <enter>
```

to receive information on all the topics on which help is available.

You can save all the variables that you have computed in a work-session by typing

```
>> save <enter>
```

before you end a work-session. This command will save all the computed variables in a file on disk named *matlab.mat*. The next time you begin a session and want to use the previously computed variables, just type

```
>> load <enter>
```

and the work-session saved in *matlab.mat* will be loaded in the current memory. You can also use an optional name of the file in which the work-session is to be saved, such as *fname.mat*, and choose selected variables (rather than all the variables to be saved), such as X, Y, Z by typing the following command:

```
>>save fname X Y Z <enter>
```

The *save* command also lets you import and export *ascii* data files.

A.2 Performing Matrix Operations in MATLAB

In the previous section we saw how MATLAB assigns and multiplies two matrices. The transpose of a matrix, **A**, is simply obtained by using the symbol ′ (*prime*) as follows:

```
>> A' <enter>
```

If **A** is a complex matrix, then **A′** is the *transpose* of the *complex-conjugate* of **A**. Adding and subtracting matrices (of the same size) is performed simply with + and - symbols, respectively:

```
>> A+B <enter>
```

or

```
>> A-B <enter>
```

If we wish to add or subtract a scalar, a, from *each element* of a matrix, **A**, then we can simply type

```
>>A+a <enter>
```

or

```
>>A-a <enter>
```

Dividing a matrix by another matrix is supported by *two matrix division* symbols / and \. The command

```
>>X=A\B <enter>
```

solves the linear algebraic equation $\mathbf{AX} = \mathbf{B}$, provided **A** is a non-singular matrix. The command

```
>>X= B/A <enter>
```

solves the linear algebraic equation $\mathbf{XA} = \mathbf{B}$, provided **A** is a non-singular matrix.

Powers of a matrix can be computed using the symbol ^ as follows:

```
>>A^p <enter>
```

where **A** is a square-matrix, and p is a scalar.

Transcendental functions of *individual elements* of matrices can be calculated using in-built MATLAB functions, such as *sin, exp, sqrt, cos, tan, asin, acos, atan, sinh, cosh, asinh, acosh, log, log10, conj, abs, real, imag, sign, angle, gcd, lcm*, etc. These commands, used in the following manner

```
>>exp(A) <enter>
```

produce a matrix whose *elements* are the required transcendental function of the *corresponding elements* of the matrix, **A**. Hence, these transcendental functions are called *array operations*, which are performed on *individual elements* of a matrix, rather than on the matrix *as a whole*. Refer to the MATLAB *Reference Guide* [2] for details on all the in-built transcendental functions available in MATLAB, or issue the *help* command. Other *array operations* are *multiplication* and *division* of the elements of one matrix by the *corresponding* elements of another matrix (of the same size), and *element-by-element powers* of a matrix. A *period* (`.`) preceding an operator (such as `*/` \ or ^) denotes an array operation. For example, the command

```
>>C = A.*B <enter>
```

denotes that elements of the matrix, **C**, are products of the elements of the matrices, **A** and **B** (both of the same size), and the command

```
>>C = A.^2 <enter>
```

denotes that the elements of the matrix, **C**, are the squares of the elements of the matrix, **A**.

Some special *matrix transcendental* functions, such as *expm*, *logm*, and *sqrtm* are also available only for square-matrices. These matrix commands have special mathematical significance, such as the *matrix exponential*, *expm*, defined in Chapter 4.

Some useful elementary matrix operations are supported by MATLAB, such as *poly* (the characteristic polynomial), *det* (determinant), *rank* (rank of a matrix), *trace* (matrix trace), *kron* (Kronecker tensor product), *inv* (matrix inverse), *eig* (eigenvalues and eigenvectors), *svd* (singular-value decomposition), *norm* (1-norm, 2-norm, F-norm, ∞-norm), *rcond* (condition number), *conv* (multiplication of two polynomials), *residue* (partial fraction expansion), *roots* (polynomial roots), etc. For a complete list and details of all the in-built matrix operations available in MATLAB, refer to the MATLAB *Reference Guide* [2], or issue the *help* command.

The *relational* operations comparing two matrices are also supported by the following MATLAB operators: < (less than), <= (less than or equal to), > (greater than), >= (greater than or equal to), == (equal to), ~= (not equal to). Comparing two matrices (of the same size) with the relational operators produces a matrix comprising 1 for each pair of elements for which the relationship is *true*, and zero for each pair of elements for which the relationship is *false*. For example, the command

```
>> [1 2; 0 5] ~= [0 2; 3 5] <enter>
```

results in the following answer:

```
ans =
    1   0
    1   0
```

MATLAB provides special functions, such as *find* and *rem*, which are very useful in relational operations. The function *find* finds the indices of the elements of a vector that satisfy a particular relational condition. When used on a matrix, *find* indexes the elements of the matrix by arranging *all the rows* in a long *column vector*, beginning with the first element of the first row, and ending with the last element of the last row. For example, if we wish to find elements of the matrix, **A**, defined in Eq. (A.1) that are greater than or equal to zero, we can simply issue the following command:

```
>> i=find(A>= 0), A(i)' <enter>

ans =
   18   9   11   26   6   0   0   7   4
```

Note that the vector **i** contains the indices of the elements of **A** that are greater than or equal to zero. Also note that we have printed-out the *transpose* of **A(i)** to save space. The function *rem* is another useful relational function. The command *rem*(A, p) produces a matrix

formed by the *remainders* of the elements of a matrix, **A**, when divided by the scalar, p. Suppose we wish to find the locations of the elements of matrix, **A**, defined in Eq. (A.1) which are exactly divisible by 3, and mark these locations by a matrix of ones and zeros, with 1 standing for each element of **A** which is divisible by 3, and 0 standing for those elements that are not divisible by 3. This is simply achieved using the following command:

```
>> rem(A,3)==0 <enter>
```

which results in the following answer:

```
ans =
   0  0  1
   1  0  0
   1  1  1
   0  1  0
```

Other useful relational functions are *isnan* (detect NaNs in a matrix), *isinf* (detect infinities in a matrix), and *finite* (detect finite values in a matrix).

The relational operations in MATLAB are based on the *logical operators*, namely & (and), | (or), ~(not). The logical operations denote *true* by 1 and *false* by 0. For example, the logical statement

```
>> ~A <enter>
```

will produce a matrix which has 1 at all locations where the corresponding elements in **A** are zeros, and 0 at all locations where **A** has non-zero elements. The logical functions *any* and *all* come in handy in many logical operations. The function *any*(**A**) produces 1 for each column of the matrix **A** that has a non-zero element, and 0 for the columns which have all zero elements. The function *all*(**A**) produces 1 for each column of the matrix **A** that has all non-zero elements, and 0 for the columns which have at least one zero element. With the matrix **A** of Eq. (A.1) the *any* and *all* commands produce the following results:

```
>>any(A) <enter>

ans =
   1  1  1

>>all(A) <enter>

ans =
   1  0  0
```

A logical function called *exist* can be used to find out whether a *variable* with a particular name exists in the work-space. For greater information on relational and logical operators, refer to the MATLAB *Reference Guide* [2].

A.3 Programming in MATLAB: Control Flow and M-Files

Instead of issuing individual MATLAB commands at the command-line, you can group a set of commands to be executed in a MATLAB *program*, called an *M-file*. The M-files have extension. *m*, and are of two types: *script files*, and *function files*. A *script file* simply executes all the commands listed in the file, and can be invoked by typing the *name* of the file and pressing <enter>. For example, a script file called *use.m* is invoked by typing *use* <enter> at the command-line prompt. After executing a script file, all the variables computed in the file are automatically stored in the work-space. The *function files* differ from *script files* in that all the variables computed inside the file are not communicated to the work-space, and only a *few* specific variables, called *input* and *output* arguments, are communicated between the *function file* and the work-space. Thus, a function file acts like a *subroutine* of a main FORTRAN, PASCAL, or BASIC program. A function file can either be called from the work-space, or from another M-file, and is therefore useful for extending the function *library* of the MATLAB. The function file must contain the word *function* at the beginning of the first line, followed by a list of *output* arguments separated by commas within *square brackets*, followed by the sign =, followed by the *name* of the function file, and finally followed by a list of *input* arguments separated by commas within *parentheses*. For example, the first line of a function file called *fred.m* looks like the following:

```
function [X, Y] = fred(A, B, C, D)
```

where A, B, C, D are *input* arguments to be specified by the *calling program* (either work-space, or another M-file), and X, Y are the *output* arguments to be returned to the calling program. The existence of this function file anywhere in the MATLAB directory defines a new MATLAB function called *fred*. All the existing MATLAB functions are, thus, in the form of function files.

The programming structure in MATLAB need not be limited to flow of information in a *sequence* of line commands. The flow of information within an M-file can be controlled using the *for* and *while* loops, and the logical *if* statements, as in any other programming language (such as DO and FOR loops, and IF statements in FORTRAN). The *for* loop in MATLAB allows a group of statements to be repeated a specified number of times. The group of statements to be repeated must end with an *end* statement. One can have nested *for* loops within *for* loops, each ending with an *end* statement. The general structure of a *for* loop is the following:

```
for i = N1:dN:N2
        statements to be repeated
end
```

where $N1$, dN, and $N2$ denote the initial value, increment, and final value of the indexing integer, i. Note that $N2$ could be *less than* $N1$, in which case dN must be *negative*.

The *while* loop allows a group of statements to be repeated an indefinite number of times, as long as a logical condition is satisfied. The general form of a *while* loop is the following:

```
while  logical expression
         statements to be repeated
end
```

The statements in a *while* loop are executed as long as the logical expression is *true* (i.e. as long as the *all* the elements of the expression matrix are *non-zero*). Usually, the expression is a scalar. An example of a *while* loop is the following:

```
A=[1 0 -1; 0 -1 1; 1 -2 1];
while norm(A, inf)<5
         A=A+0.1;
end
```

where *norm* (A, *inf*) denotes the *infinity* norm of the matrix, **A**.

The *if* and *else* statements allow a group of statements to be executed, if a specified logical expression is *true*, and a *second* group of statements to be executed, if the *same* logical expression is *false*. It is also possible to execute a *third* set of statements, if the specified logical expression in *false*, and *another* logical expression is *true*, using the statement *elseif*. Each *if*, *else*, *elseif* block of statements must be followed by the *end* statement. For example, the following program illustrates how a computation can be carried out in three cases, depending upon the value of a scalar, p:

```
if p<1.0
         A=B*C;
elseif p==1.0
         A=(B.*B)*C;
else
         A=zeros(size(B*C));
end
```

Using the *if* statement, it is possible to come out of a *for* or *while* loop with the *break* statement. For example, if you *do not* wish to repeat a *while* loop *more* than 100 times, you can use the *if* and *break* combination as follows:

```
  n=0;
  while  expression
  n=n+1
  if n>100, break, end
  statements to be repeated
end
```

You can create your own online help for the M-files you have programmed by adding *comment statements* immediately after the first line of the file. A comment statement begins with the symbol % and are not executed by MATLAB. Some programs may

require *strings* of texts for their execution. A string of text is specified by entering text within single quotes, such as:

```
>>g = 'goodbye' <enter>
```

which results in

```
s =
     goodbye
```

A mathematical expression can be included as a text string, and you can use the function *eval* to evaluate the value of the expression.

There are several advanced ways of providing *input* and *output* data to and from MATLAB, such as using a shell escape to an externally running program, importing and exporting data using *flat files, MEX-files*, and *MAT-files*, or with the MATLAB functions *fopen, fread*, and *fwrite* for disk data files. For further information on data transfer to and from MATLAB, refer to the MATLAB *User's Guide* [1].

Programming in MATLAB is made easy with the availability of the *command-line editor*, which displays error messages if a command is incorrectly used, or with the help of *debugging* commands, such as *dbstop, dbclear, dbcont, dbstack, dbstatus*, etc. A simple way of checking whether your M-file is doing what it is supposed to do, is displaying the results of selected intermediate computations by removing semi-colons at the end of selected statements.

Finally, you can post-process your computations by plotting important variables using MATLAB's extensive graphical capabilities. The most commonly used MATLAB graphical commands are *plot*(X, Y) (for generating a plot of the elements of vector, $\mathbf{Y}$, against the vector, $\mathbf{X}$). Most of the graphs contained in this book have been generated using the *plot* command. You should carefully study the various options available in executing the plot command [1], and also other graphical commands, such as *semilogx* (a plot with a log scale on the x-axis), *semilogy* (a plot with a log scale on the y-axis), *loglog* (a plot with log scales on both x- and y-axes), *subplot* (for displaying more than one plots at a time), *grid* (for generating a grid for a plot), etc. Other MATLAB 2-D graphical functions include *stairs* (staircase plot), *bar* (bar-chart), *hist* (histogram), *feather* (feather plot for angles and magnitudes of complex numbers), *polar* (plot in polar coordinates), *quiver* (plots of vector magnitudes and directions), *rose* (angle histogram), *fill* (solid polygonal plot), and *fplot* (plot of an evaluated mathematical function). There are also a range of 3D plotting functions available in MATLAB. Refer to the *User's Guide* [1] for details on graphical functions.

A.4 The Control System Toolbox

The *Control System Toolbox* (CST) for use with MATLAB provides *additional* function M-files (apart from the basic MATLAB functions) that are especially useful in the analysis and design of control systems. Most of the function files from CST have been extensively used throughout this book, and you have been provided information on how to invoke

the associated commands in the main text. For additional information about the CST functions, such as *bode, dbode, are, c2d, c2dm, damp, ddamp, nyquist, dnyquist, lsim, dlsim, tf, ss, estim, destim, lqr, dlqr, lqe, dlqe, sigma, dsigma, place, acker, ngrid, nichols, reg, dreg, initial, dinitial, step, dstep, impulse, dimpulse, series, parallel, feedback, margin, rlocus*, etc., you may refer to the *User's Guide for Control System Toolbox* [3], or issue the *help* command. Two valuable user-friendly graphical tools are also available with CST: the *LTI Viewer*, which lets you view all necessary information required in analyzing a linear, time-invariant system at the click of the mouse button, and the *SISO design tool*, which leads you step-by-step in the graphical window world of designing single-input, single-output, LTI systems. To access these tools, go to the MATLAB *launch pad*, click on the + sign next to the Control System Toolbox, and select any of the two tools that appear on the selection tree. As you become sufficiently proficient with MATLAB programming, you may find it relatively easier to write your own function files for carrying out many of the control analysis and design tasks detailed in CST, using the basic MATLAB functions and the theoretical background in control systems provided in this book. Some examples of the new function M-files have been listed elsewhere in this book. However, for a beginner in controls, the CST function files are valuable tools for learning the tricks of the trade. Apart from the CST, there are several other toolboxes available for advanced control applications, such as the *Signal Processing Toolbox, System Identification Toolbox, Optimization Toolbox, Robust Control Toolbox, Nonlinear Control Design Toolbox, Neural Network Toolbox*, and *μ-Analysis and Synthesis Toolbox*. As your control applications become advanced, you may wish to add some of these advanced toolboxes to your MATLAB directory. Information on how to order these toolboxes can be obtained from the MathWorks, Inc., 24 Prime Park Way, Natick, MA.

A.5 SIMULINK

SIMULINK is a Graphical User's Interface (GUI) software which works directly with the block-diagram of a control system (rather than differential equations, or transfer functions) to produce a simulation of the system's response to arbitrary inputs and initial conditions. The basic entity in SIMULINK is a *block*, which can be selected from a *library* of commonly used blocks. Alternatively, a user can devise special blocks out of the common blocks, M-files, MEX files, C, or Java-codes through the *S-function* facility. The procedure for carrying out a system's simulation through SIMULINK is the following:

1. Double click on the SIMULINK icon on the MATLAB toolbar, or issue the command *simulink* < enter > at the MATLAB prompt (>>). The SIMULINK *library browser* window will open.
2. Click on the *create a new model* icon on the SIMULINK *toolbar*. A window for the new model will open.
3. Open the *subsystem* library in the general SIMULINK *library browser* by double-clicking on the appropriate icon. The subsystems are: *continuous, discrete, functions & tables, math, nonlinear, signals & systems, sinks*, and *sources*.

4. Select the required blocks from the *subsystems* libraries, and drag them individually to the open *new model* window.

5. Once you have dragged the required blocks to the *new model* window, you can join the *in-ports* and *out-ports* of the adjacent blocks to create a block-diagram as desired. You can double-click on each block in your model to open a *dialog box*, in which the block's *parameters* can be set.

6. Once the block-diagram is complete, you can save it using the *save* button on the new model's toolbar.

7. Now you are ready to begin the simulation of your control system. Just go to the toolbar of the model you have saved and click on the *play* button. If you have created your model correctly, the simulation will start and you can view the results using any of the *sink* blocks in your model. However, one seldom succeeds at first, and SIMULINK prompts you through a *diagnostics dialog box* to tell you what went wrong with the simulation, and also what you should modify in your model for a successful simulation.

8. You can refine your simulation by adjusting the *simulation parameters* that drop down when you click on the *simulation* button on the model's toolbar.

Following the above steps, any practical control system can be simulated accurately using SIMULINK. Let us briefly see the contents of each SIMULINK *subsystem block library*.

Continuous: *transfer function, state-space, integrator, derivative, transport delay, variable transport-delay, memory, zero-pole.* (These block help you construct a continuous-time (analog) system model.)

Discrete: *discrete transfer function, discrete state-space, discrete zero-pole, discrete filter, discrete-time integrator, first-order hold, zero-order hold, unit delay.* (These block help you construct a discrete-time (digital) system model.)

Functions & Tables: this library contains specialized functions and tables blocks useful for creating complicated systems. It includes all MATLAB intrinsic functions, as well as special user created functions through the *S-function* block.

Math: all the mathematical connection blocks (such as *sum junction, gain, matrix gain, product, dot product, abs, floor, trigonometric function*, etc.), and relational and logical operator blocks (such as *and, combinatorial logic*, etc.) are found here. These blocks are indispensable in constructing any control system.

Nonlinear: contains a number of nonlinear system blocks that are very useful in modeling a variety of physical phenomena. Some examples are *backlash, coulomb & viscous friction, dead zone, saturation, rate limiter, relay, switch,* etc.

Signals & Systems: this library contains many specialized blocks used for representing subsystems and operating on signals passing through a system. Some commonly useful

blocks are *subsystem* (which allows you to group a number of blocks into a subsystem), *mux, demux, in1,* and *out1.*

Sinks: contains a number of possible ways of output of data from a model, such as *scope, xy graph, to workspace, simout, display,* and *stop simulation.* For example, a *scope* can be used to directly view a simulation variable in a window of the model, and to also store the data in a file.

Sources: provides a variety of input sources for the model, such as *step, ramp, sine wave, pulse generator, random number, repeating sequence, band-limited white noise, chirp signal, signal generator,* etc.

The SIMULINK provides several *simulation parameters* that can be adjusted to achieve a desired accuracy in a simulation. A user can select from a number of time-integration schemes, such as *Runge–Kutta, Adams, Euler, predictor-corrector,* as well as refine the tolerances and time step sizes used for performing the simulation. Useful diagnostics are generated to let a user improve her simulation.

The most useful feature of SIMULINK is that you can use variables specified in the MATLAB work-space as block parameters, and in this manner work seamlessly with all the intrinsic and toolbox functions of MATLAB. For more information on SIMULINK, refer to its user's guide [4], or work interactively with the SIMULINK blocks and models until you get a hang of it. Once understood, SIMULINK modeling can become a powerful tool in the hands of a control systems designer.

References

1. *MATLAB 6.0 User's Guide.* The Math Works Inc., Natick, MA, USA, 2000.
2. *MATLAB 6.0 Reference Guide.* The Math Works Inc., Natick, MA, USA, 2000.
3. *Control System Toolbox 5.0 for Use with MATLAB-User's Guide.* The Math Works Inc., Natick, MA, USA, 2000.
4. *SIMULINK 4.0 User's Guide.* The Math Works Inc., Natick, MA, USA, 2000.