System Dynamics (22.554 & 24.509)

V. Modeling and Simulating Dynamic Systems with Matlab/Simulink

Introduction

We have already seen some of the capabilities of Matlab for matrix manipulation and linear algebra operations. In addition, some simple time-domain dynamic simulations of low-order state-space models have been performed. Although our primary focus is on the general state-space matrix formulation, several alternate representations are possible, and Matlab makes it easy to convert between various representations. It also has functions for simulation and plotting in both the time and frequency domains. A summary of the various representations of linear time invariant systems within Matlab is given in Fig. 5.1. Note that both discrete and continuous systems can be modeled with analogous treatments, and that the conversion of continuous to discrete form (or vice versa) is possible at the state space level. In this section of notes, many of the operations indicated in Fig. 5.1 for continuous systems are illustrated via examples. The reader should see Tables 5.1 and 5.2 at the end of this section for a series of three sample input-output sessions in Matlab. These tables contain listings of **modeldemo1/2/3.m** and **modeldemo1/2/3.out**, respectively (the diary files have been edited slightly for presentation here). The physical model treated in this example is described below.

Consider, for example, a simple mass-spring-dashpot mechanical system that can be described by the following force balance,

$$m\frac{d^2}{dt^2}y(t) + c\frac{d}{dt}y(t) + ky(t) = u(t) \tag{5.1}$$

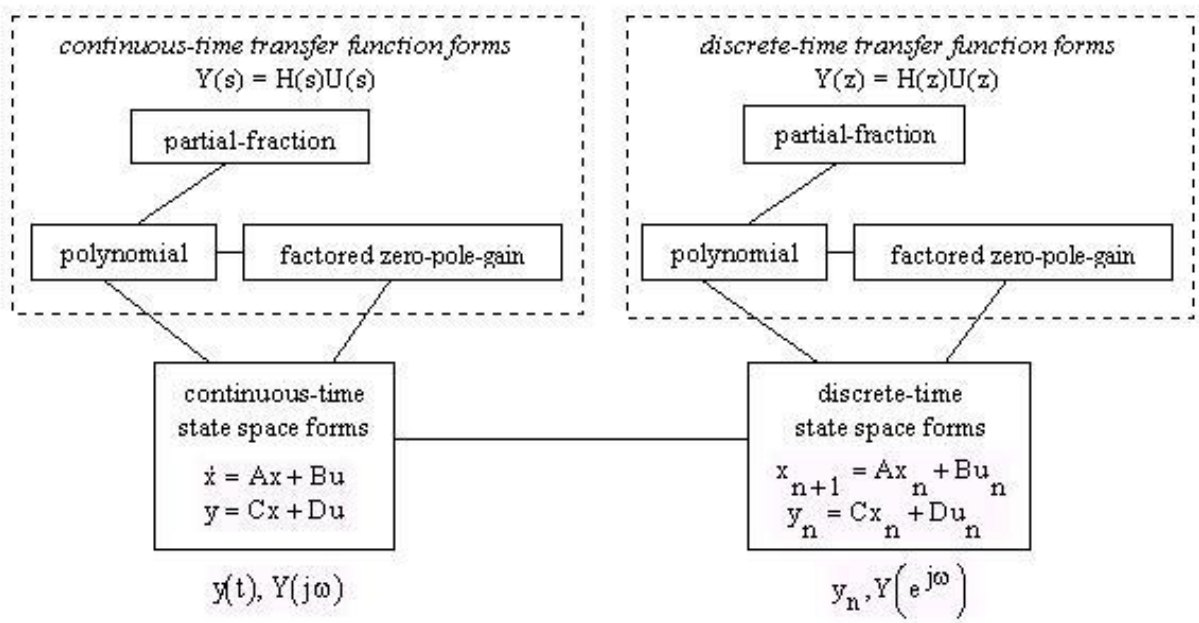


Fig. 5.1 Model conversion relationships in Matlab.

where y(t) is the instantaneous displacement of the system from equilibrium, and k and c are proportionality constants that describe the spring and dashpot properties, respectively. As we have seen in previous assignments, this system can be put into state matrix form by letting $x_1 = y$ and $x_2 = dy/dt$, giving

$$\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -k/m & -c/m \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1/m \end{bmatrix} u(t) \quad \text{and} \quad y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{5.2}$$

where the $\underline{\underline{A}}$, $\underline{\underline{B}}$, $\underline{\underline{C}}$, and $\underline{\underline{D}}$ matrices associated with the standard state space format for linear time invariant systems are given by direct comparison with the general equations,

$$\frac{d}{dt}\underline{x} = \underline{\underline{A}}\,\underline{x} + \underline{\underline{B}}\,\underline{u} \quad \text{and} \quad \underline{y} = \underline{\underline{C}}\,\underline{x} + \underline{\underline{D}}\,\underline{u} \tag{5.3}$$

Time Domain Simulation

Once the defining matrices are available and an LTI object has been defined, time domain simulations are particularly simple using three different Matlab functions, **impulse**, **step**, and **lsim**. These functions for the simulation of a state space system have already been introduced in Section III. For example, the sequence for an arbitrary input function could be written as

sys = ss(A,B,C,D), [Y,T,X] = lsim(sys,U,t,xo)

There are special forms that can be used in certain applications and the requirements on the format of certain elements (such as the input **U** matrix in the **lsim** function) may not be completely obvious, so careful use of the functions is required. Also note that the **impulse**, **step**, and **lsim** commands can be used with the transfer function form of a system [where the LTI object is defined via **sys = tf(num,den)** -- as already discussed in Section IV]. Therefore, in general, because of the number of possibilities that exist, it is good practice for the user to consult Matlab's help files for a complete description of each function. As a typical example, the **impulse** and **step** functions are used to show the time domain behavior of the mass-spring-dashpot system modeled in the Matlab sample session at the end of this section (see Figs. 5.2 and 5.3 and Tables 5.1 and 5.2).

Model Conversion

The transfer function representation of systems has also been described in some detail. In the state-space formulation, one has

$$\underline{Y}(s) = \underline{\underline{G}}(s)\underline{U}(s) \tag{5.4}$$

where the system transfer function matrix is given as

$$\underline{\underline{G}}(s) = \underline{\underline{C}}\left[s\underline{\underline{I}} - \underline{\underline{A}}\right]^{-1}\underline{\underline{B}} + \underline{\underline{D}} \tag{5.5}$$

Performing the indicated matrix operations for the single input-single output (SISO) mechanical system described above gives a single scalar transfer function,

$$G(s) = \frac{1}{ms^2 + cs + k} \tag{5.6}$$

Note that this could also be easily derived from the original second-order differential equation. In general, however, multiple input-multiple output (MIMO) systems are quite tedious to work with algebraically, and the model conversion functions in Matlab are quite useful in this regard.

In particular, the conversions from state-space form to transfer function form (and vice versa) and from the transfer function form to zero-pole-gain form are very useful. With the LTI object representation in Matlab, these conversions are quite straightforward. The **ss**, **tf**, and **zpk** commands easily convert among the various forms. The Matlab functions to do the conversions suggested above are

sys1 = ss(A,B,C,D), sys2 = tf(sys1), sys3 = zpk(sys2)

One can also extract the information stored within the data structure associated with a particular LTI object with the **ssdata**, **tfdata**, and **zpkdata** commands, or by direct structure-like referencing. For example, the command, **[num,den] = tfdata(sys2)**, returns the numerators and denominators of the LTI object **sys2**. **num** and **den** are cell arrays with as many rows as outputs and as many columns as inputs, and their i,j entries specify the transfer function from input j to output i. Each element of the cell array is a vector that contains the coefficients of the numerator or denominator polynomial in s, in order of decreasing powers of s. Note that the **sys2** object is first converted to a transfer function object if necessary.

An alternate approach would be to extract the individual cell arrays directly from the data structure and then display it using Matlab's **celldisp** command and/or perform other manipulations as desired. The sequence for simply displaying the data could be accomplished with the following commands:

```
fieldnames(sys2)
num1 = sys2.num,   den1 = sys2.den
celldisp(num1),   celldisp(den1)
```

Note also that, if we desire to actually extract the numerical data from the cell array into a numeric variable, we need to use the curly brackets, { }, within the command. For example, to extract the numerical data for the 1,1 element of the transfer function matrix, we have

nnum1 = num1{1,1}, nden1 = den1{1,1}

where **num1** and **den1** are cells arrays (from the above example) and **nnum1** and **nden1** are numeric arrays containing the coefficients of the numerator and denominator polynomials for the desired element of the transfer function matrix. Note that using **cden1 = den1(1,1)** returns a cell array in the **cden1** variable. Thus, the use of curly brackets, { }, returns numeric values, and the use of the normal parentheses, ( ), returns a cell structure when addressing a particular element of a cell array.

Similar manipulations can be performed with the zero-pole-gain representation. In particular, the command, **[Z,P,K] = zpkdata(sys3),** returns the zeros, poles, and gain for each IO channel of the LTI system **sys3**. The **Z** and **P** cell arrays and the **K** matrix have as many rows as outputs and as many columns as inputs, and their i,j entries specify the zeros, poles, and gain of the transfer function from input j to output i.

In the case of the single-input single-output system (SISO) given in the current example, the transfer function and zero-pole-gain form simplify to the common representation given by

$$G(s) \Rightarrow \frac{\text{num}(s)}{\text{den}(s)} \Rightarrow K\frac{(s-z_1)(s-z_2)\cdots}{(s-p_1)(s-p_2)(s-p_3)\cdots} \tag{5.7}$$

The help files for the various Matlab commands mentioned above should be consulted for further details in using these functions and also on the several related functions that are available. Also the user should be cautioned that model conversion for large systems is a tricky business because of potential numerical difficulties -- so make sure that the results are reasonable before continuing. The Matlab sample session listed in Tables 5.1 and 5.2 illustrates several conversion routines for the current SISO mechanical system example.

Residue or Partial Fraction Form

In many cases, one may also be interested in reducing a ratio of polynomials in s to the partial fraction expansion form. For example, G(s) for the SISO mechanical system can be written as

$$G(s) = \frac{B(s)}{A(s)} = \frac{1/m}{s^2 + \frac{c}{m}s + \frac{k}{m}} = \frac{r_1}{s-p_1} + \frac{r_2}{s-p_2} + k(s) \tag{5.8}$$

where, in this case,

$$p_{1,2} = \frac{-c \pm \sqrt{c^2 - 4mk}}{2m} \tag{5.9}$$

and r_1 and r_2 can be computed using a variety of methods [and k(s) = 0]. For problems of this type, Matlab has the **residue** function, **[r,p,k] = residue(B,A)**, where **B** and **A** are row vectors containing the polynomial coefficients and **r** and **p** are column vectors containing the residues and poles, respectively. **k(s)** is only defined if the order of **B(s)** is greater than that of **A(s)**, which is not the usual case for the analysis of linear systems.

Frequency Domain Simulation

Performing frequency domain simulations is conceptually straightforward, but they can be computationally intensive. In practice, one simply evaluates the expression,

$$\underline{\underline{G}}(j\omega) = \underline{\underline{C}}\left[j\omega\underline{\underline{I}} - \underline{\underline{A}}\right]^{-1}\underline{\underline{B}} + \underline{\underline{D}} \tag{5.10}$$

for several values of ω and then usually plots the data in one of three ways -- using Bode, Nichols, or Nyquist plots. In Matlab, some preconditioning of the matrices is performed for increased computational efficiency, but this work is transparent to the user.

Matlab has several functions for obtaining frequency domain signatures. The first step is to generate a vector of frequencies. This is usually done so that the points are logarithmically spaced between decades 10^{d1} and 10^{d2} with a total of n points with **w = logspace(d1,d2,n)**. The **bode** or **nyquist** functions can then be used to evaluate $\underline{\underline{G}}(j\omega)$ at each frequency point,

[MAG,PHASE] = bode(sys,w)

or

[RE,IM] = nyquist(sys,w)

If **sys** has NU inputs and NY outputs and LW = length(**w**), **MAG** and **PHASE** are NY×NU×LW arrays and the response at the frequency **w(k)** is given by **MAG(:,:,k)** and **PHASE(:,:,k)**. A similar description of the **nyquist** output can be obtained from the Matlab help files. The **MAG** variable is the magnitude of the transfer function in absolute units, and the Matlab command **MAGDB = 20*log10(MAG)** converts it into decibels.

Finally, with the frequency response data available, one can construct various frequency response plots using Matlab's standard plot functions. Also note that Matlab will automatically generate the desired plots if there are no left hand side arguments in the function call. This often gives a better plot with a lot less work. For example, the frequency response for the SISO mass-spring-dashpot system is computed and graphed as part of the Matlab sample session given in Table 5.1. A comparison of the time domain responses in Figs. 5.2 and 5.3 (impulse and step responses) and the various frequency domain signatures in Figs. 5.4 to 5.6 (Bode, Nichols, and Nyquist plots) can help identify characteristic features that are common for typical second-order systems.

Block Diagrams/Model Building

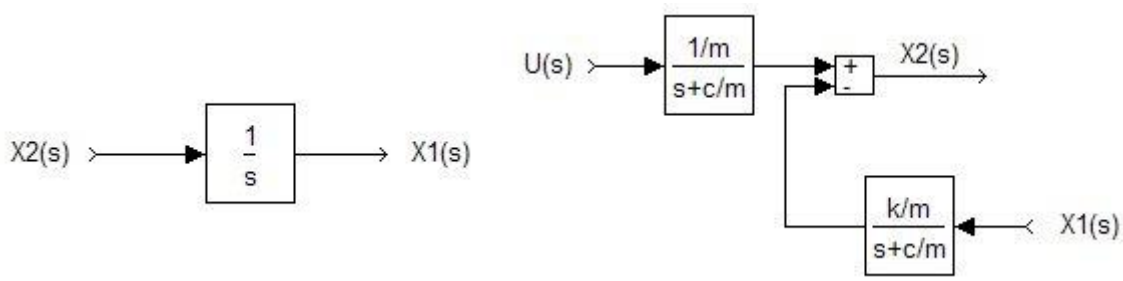
Many real systems that are comprised of several components and feedback loops are given in block diagram form. It can often be quite tedious to find equivalent system representations, $\underline{\underline{G}}(s)$, and the corresponding state-space matrices, $\underline{\underline{A}}$, $\underline{\underline{B}}$, $\underline{\underline{C}}$, and $\underline{\underline{D}}$ via hand manipulation. But, as usual, we can use Matlab to help build the state-space representation from the block diagram form (although this is really a rather advanced feature of the Matlab system and numerical difficulties can occur). One can do this in Matlab using a variety of commands within the Control Toolbox or you can use the Simulink graphical interface for modeling systems.

To illustrate these processes, we will work with the simple mechanical system example given above. Let's first put this system in block diagram form, and then we can automatically recreate the original state space and overall transfer function forms as a test of the model building capabilities in Matlab and Simulink. Expanding the state-space equations and taking the Laplace transform of the individual equations gives

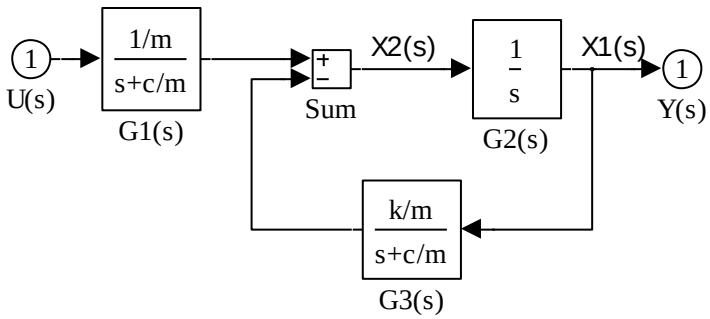
$$\frac{d}{dt}x_1 = x_2 \qquad\qquad X_1(s) = \frac{1}{s}X_2(s)$$

$$\frac{d}{dt}x_2 = -\frac{k}{m}x_1 - \frac{c}{m}x_2 + \frac{1}{m}u \qquad\qquad X_2(s) = \frac{-k/m}{s+c/m}X_1(s) + \frac{1/m}{s+c/m}U(s)$$

From the Laplace transform equations, it is easy to represent individual blocks, as follows:



Now connecting these blocks and noting that the desired output is $Y(s) = X_1(s)$, gives



Some minimal block diagram arithmetic (a feedback part in series with the first block) gives the overall transfer function between Y(s) and U(s), where the resultant G(s) is the same as before [see eqn. (5.6)].

The Old Way (using command-based code)

Now the challenge is to get Matlab to take this overall system block diagram and automatically generate the state space and global transfer function representations. The Control Toolbox has a number of functions that can be used to accomplish this task. Although each case can be treated differently, there are five basic steps in this procedure, as follows:

1. The first step is to define the individual transfer functions for each block (numbered 1, 2, 3, ..., nblocks). In this simple example we have three blocks where each transfer function is defined as the ratio Nn(s)/Dn(s). For this case one can define the following transfer functions (see Tables 5.1 and 5.2 for the actual Matlab representation):

   $$n1 = 1/m \quad \& \quad d1 = s + c/m \qquad n2 = 1 \quad \& \quad d2 = s \qquad n3 = k/m \quad \& \quad d3 = s + c/m$$

2. We then build an unconnected state-space model by repeated calls to Matlab's **tf** and **append** commands, as needed. For example, we can append blocks 1, 2 and 3 to give an 'unconnected' system with the following commands:

   sys1 = tf(n1,d1), sys2 = tf(n2,d2), sys3 = tf(n3,d3)
   sysuct = append(sys1,sys2,sys3), sysucs = ss(sysuct)

   which gives the system

   $$\frac{d}{dt}\begin{bmatrix} \underline{x}_1 \\ \underline{x}_2 \\ \underline{x}_3 \end{bmatrix} = \begin{bmatrix} \underline{\underline{A}}_1 & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{A}}_2 & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{A}}_3 \end{bmatrix}\begin{bmatrix} \underline{x}_1 \\ \underline{x}_2 \\ \underline{x}_3 \end{bmatrix} + \begin{bmatrix} \underline{\underline{B}}_1 & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{B}}_2 & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{B}}_3 \end{bmatrix}\begin{bmatrix} \underline{u}_1 \\ \underline{u}_2 \\ \underline{u}_3 \end{bmatrix}$$

   $$\begin{bmatrix} \underline{y}_1 \\ \underline{y}_2 \\ \underline{y}_3 \end{bmatrix} = \begin{bmatrix} \underline{\underline{C}}_1 & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{C}}_2 & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{C}}_3 \end{bmatrix}\begin{bmatrix} \underline{x}_1 \\ \underline{x}_2 \\ \underline{x}_3 \end{bmatrix} + \begin{bmatrix} \underline{\underline{D}}_1 & \underline{\underline{0}} & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{D}}_2 & \underline{\underline{0}} \\ \underline{\underline{0}} & \underline{\underline{0}} & \underline{\underline{D}}_3 \end{bmatrix}\begin{bmatrix} \underline{u}_1 \\ \underline{u}_2 \\ \underline{u}_3 \end{bmatrix}$$

   To simplify this process for the case of several system blocks, the **append** command can have as many systems in the input argument list as needed.

3. The next step is to define a matrix, **Q**, which indicates how the individual blocks are interconnected. The matrix **Q** has a row for each block. The first element in each row is the block number and the subsequent entries contain the block numbers where the current block gets its summing inputs (a negative block number implies a negative feedback connection). Zeros are used as needed to fill the array so that each row has the same number of columns. For the above example, the **Q** matrix would be given as

   Q = [1 0 0; 2 1 -3; 3 2 0]

4. Identify the system inputs and outputs in row vectors **iu** and **iy**, respectively, where the entries are simply the appropriate block numbers. In our example of a simple mass-spring-dashpot system, we only have a single input [i.e. u(t)] and a single output [i.e. $y(t) = x_1(t)$]. For this case the **iu** and **iy** vectors are identified as

   iu = [1], iy = [2]

5. The final step is to simply connect the blocks as described in the **Q** matrix into a full state matrix representation. The appropriate statement is

   syscs = connect(sysucs,Q,iu,iy)

Performing these steps gives a state matrix formulation equivalent to the system described via the original block diagram. Thus, the model building process is complete.

In many cases the above process does not give a minimal state space realization of the system. This implies that there are probably some poles and zeros that can be canceled. Generally, this is not a problem, but minimal systems are usually more efficient during time and frequency domain simulations. Matlab has commands to remove the extra states and find a minimal realization (see **minreal**). For example, for this system the proper command is

syscsm = minreal(syscs)

At this point the user has an overall state-space formulation of the original system that was in the form of several connected MIMO blocks. The block diagram arithmetic that can often be a very tedious undertaking has been completed automatically, resulting in a set of state-space matrices that capture the full dynamics of the original system. One can now work with this system as desired. In **modeldemo2.m** and **modeldemo2.out** (see Tables 5.1 and 5.2), we simply show that conversion back into the transfer function form gives the original transfer function that we started with; thus successfully completing this modeling demonstration. Also note that numerical difficulties can be encountered in the model conversion process. We did not have any difficulty with this particular example but, in general, the user should be careful to systematically evaluate and study the resultant system behavior.

The New Way (using the Simulink graphical interface)

As a final task in this Matlab modeling demonstration, let's build the same system as just described, but this time we will use the Simulink graphical interface. In simple terms, Simulink represents a graphical interface which automatically performs the **append** and **connect** operations that we illustrated above. In fact, however, Simulink has evolved into a very powerful modeling capability that can perform sophisticated analyses of coupled linear and nonlinear systems within a relatively easy-to-use graphical environment.

We will briefly explore some additional Simulink capabilities at a later time. For now, the goal is to simply build the above 3-block system within Simulink, extract the $\underline{\underline{A}}$, $\underline{\underline{B}}$, $\underline{\underline{C}}$, and $\underline{\underline{D}}$ state-space matrices from the resultant model, and illustrate that it gives the expected dynamic response for this simple mechanical spring-mass-dashpot system.

The process starts with the graphical representation of the system in Simulink. After typing **simulink** at the Matlab command line and opening a new model file, one simply drags the appropriate blocks that are needed to represent a given system from the Simulink block library to the workspace area. Dragging the mouse between desired blocks connects the inputs and outputs of each block. Finally, the individual blocks are customized to contain the desired parameters for the current simulation. The resulting graphical representation is saved as a model file (with an **slx** or **mdl** extension) for later use within Matlab or as a standalone Simulink model.

In the current demo, the Simulink model was saved as **modeldemo3_sl.slx** (see footnote). Simply typing **modeldemo3_sl** at the Matlab prompt displays the graphical representation of the system (see block diagram at the top of page 6). Also, as the last step in this series of demonstrations, **modeldemo3.m** gives an illustration of how to use the graphical model embedded in **modeldemo3_sl.slx**. In this case we simply extract the $\underline{\underline{A}}$, $\underline{\underline{B}}$, $\underline{\underline{C}}$, and $\underline{\underline{D}}$ LTI state matrices using the **linmod** command,

[A,B,C,D] = linmod('modeldemo3_sl')

and proceed to show that the system is identical to those generated above (see the code and diary output listings in Tables 5.1 and 5.2 for further details). Thus, this first Simulink demo simply shows that the Simulink graphical interface can be used for model construction as an alternative to the **append**, **connect**, **…** sequence from the Control Toolbox library. Simulink, however, is often the preferred option primarily because of its wide range of capabilities, overall flexibility, and general ease of use.

We have now completed this brief demo illustrating some of Matlab's modeling capabilities. In general, the reader is encouraged to browse through the help files for many of Matlab's interesting and useful functions and to give Simulink a try. With some experience, you will find that the Matlab/Simulink combination is one of the best tools available for the modeling and simulation of medium-sized dynamic systems. Feel free to experiment with some of the many features these codes have to offer. Also, be sure to take advantage of their capabilities in some of your other courses, as appropriate...

Note: The recent versions of Simulink allow saving the model file as a ***.slx** binary file or as a ***.mdl** ascii file, with the ***.slx** format being the default and more efficient file type. These files store all the necessary information so that Simulink can regenerate the system model within its graphical user interface. Most of the information in these files is not particularly useful to the general user, but you never know -- however, with the ascii-formatted ***.mdl** file format, it is easily accessible if you need it for some reason…

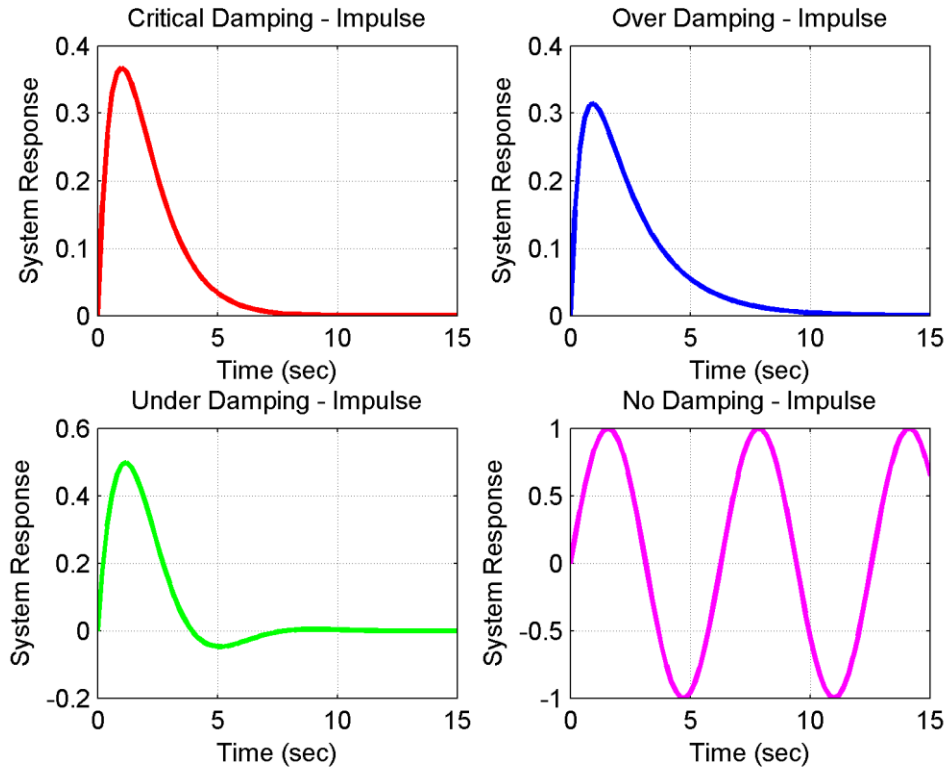


Fig. 5.2 Impulse response of typical second order system.

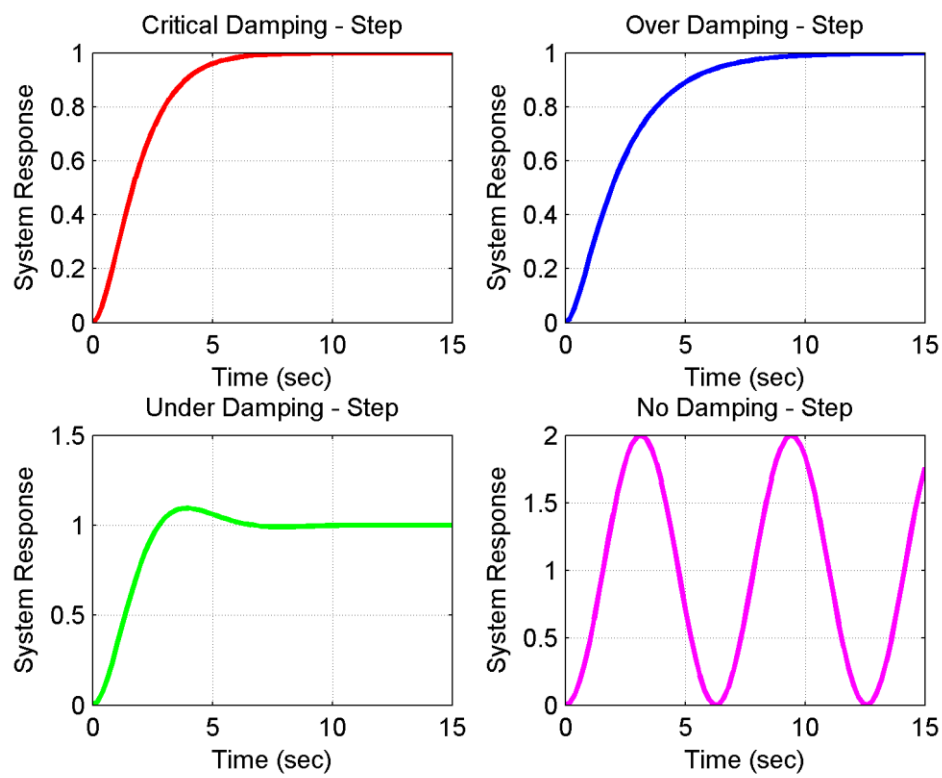


Fig. 5.3 Step response of typical second order system.

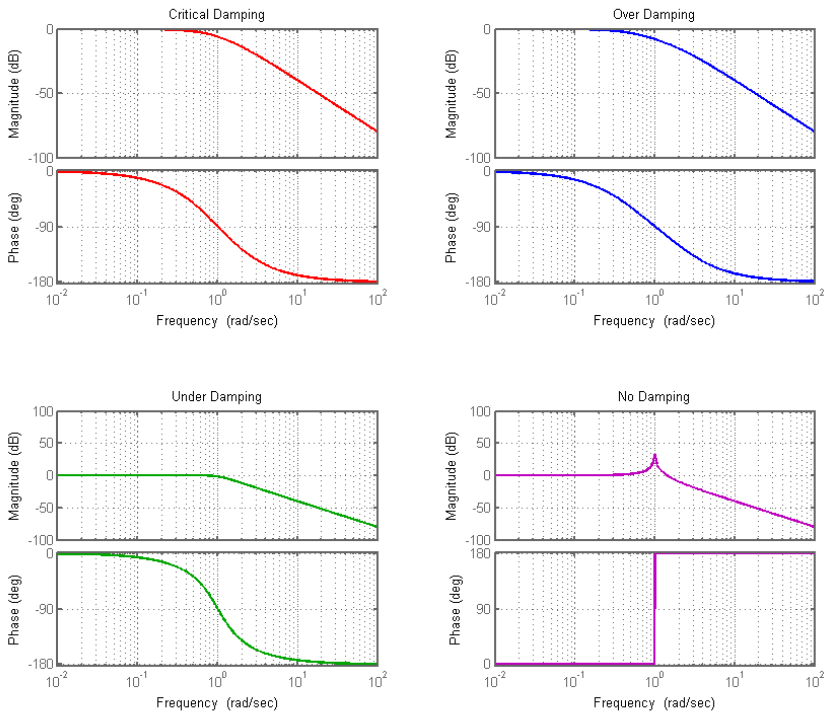


Fig. 5.4 Bode plots for typical second order system.

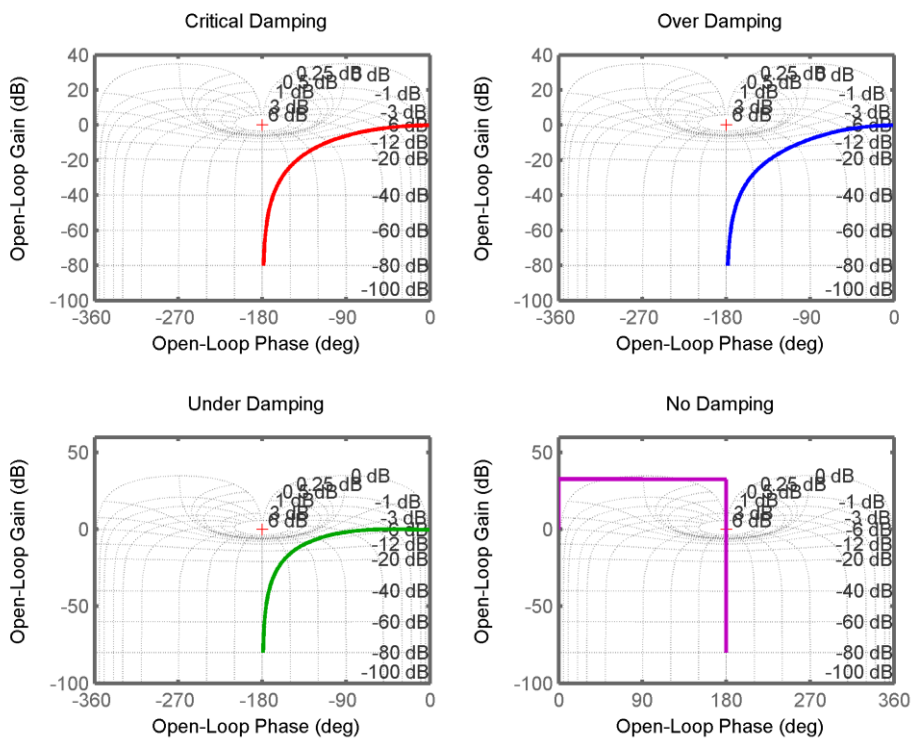


Fig. 5.5 Nichols plots for typical second order system.

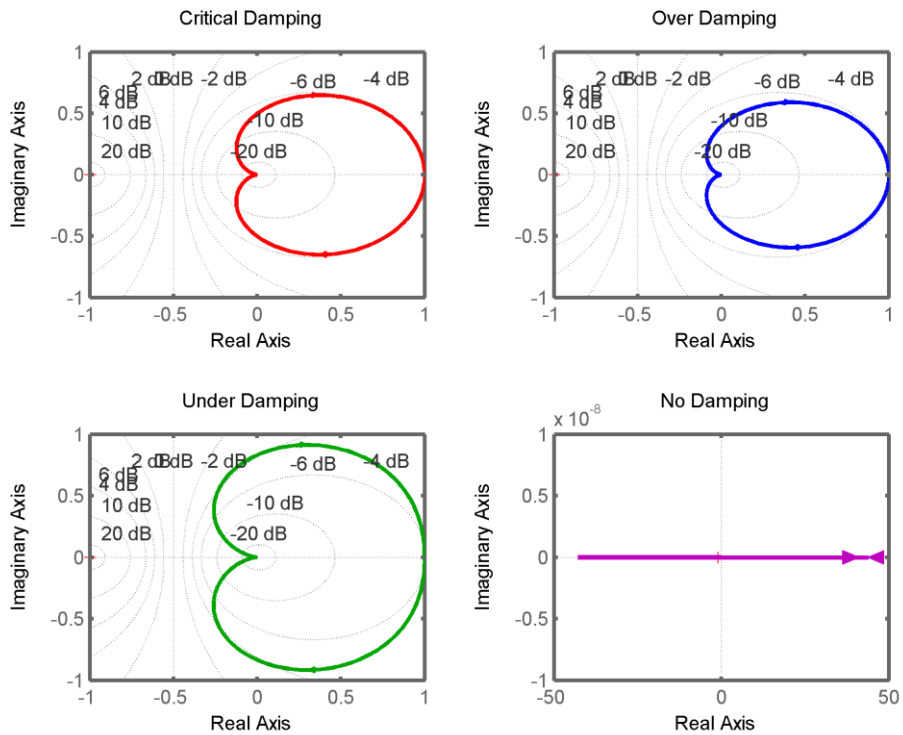


Fig. 5.6 Nyquist plots for typical second order system.

Table 5.1 Listing of Matlab files modeldemo1/2/3.m.

```
%
%   MODELDEMO.M   Sequence of Matlab sample programs that demonstrate
%               various modeling and simulation techniques (both time and
%               frequency domain) including a simple Simulink sample
%
%   Mass-Spring-Dashpot Mechanical System
%     my''(t) + cy'(t) + ky(t)  =  u(t)
%
%   This example has been broken into three parts (3 files), as follows:
%    Demo 1: simple simulation and model conversion techniques    (*** this file ***)
%    Demo 2: modeling of systems in block diagram form (the OLD way)
%    Demo 3: modeling with Simulink (the NEW way)
%
%   File prepared by J. R. White, UMass-Lowell (last update: Feb. 2020)
%

      clear all, close all, nfig = 0;
      format compact
%
%   open diary file for saving solutions
      delete modeldemo1.out,    diary modeldemo1.out
      disp('  *** MODELDEMO1.OUT  ***   Diary File for MODELDEMO1.M '),   disp(' ')
%
%   define coefficients for several cases
      m = 1                    % kg      mass
      k = 1                    % kg/s^2  spring const
      c = [2.0 2.5 1.2 0.0]    % kg/s    critical/over/under/no damping factor
%
%   create state space model for each case
```

```
      disp('  State Space Models')
      km = k/m;
      A1 = [0 1;-km -c(1)/m],     A2 = [0 1;-km -c(2)/m]
      A3 = [0 1;-km -c(3)/m],     A4 = [0 1;-km -c(4)/m]
      B = [0 1/m]',     C = [1 0],     D = [0]
      sys1s = ss(A1,B,C,D);     sys2s = ss(A2,B,C,D);
      sys3s = ss(A3,B,C,D);     sys4s = ss(A4,B,C,D);
%
%   simulate different time domain responses (impulse & step inputs)
      t = 0:.2:15;            % time vector (seconds)
%
      y1 = impulse(sys1s,t);  y2 = impulse(sys2s,t);
      y3 = impulse(sys3s,t);  y4 = impulse(sys4s,t);
      nfig = nfig+1;  figure(nfig)
      subplot(221),plot(t,y1,'r','LineWidth',2),title('Critical Damping - Impulse'),grid
      xlabel('Time (sec)'),ylabel('System Response')
      subplot(222),plot(t,y2,'b','LineWidth',2),title('Over Damping - Impulse'),grid
      xlabel('Time (sec)'),ylabel('System Response')
      subplot(223),plot(t,y3,'g','LineWidth',2),title('Under Damping - Impulse'),grid
      xlabel('Time (sec)'),ylabel('System Response')
      subplot(224),plot(t,y4,'m','LineWidth',2),title('No Damping - Impulse'),grid
      xlabel('Time (sec)'),ylabel('System Response')
%
      y1 = step(sys1s,t);  y2 = step(sys2s,t);
      y3 = step(sys3s,t);  y4 = step(sys4s,t);
      nfig = nfig+1;  figure(nfig)
      subplot(221),plot(t,y1,'r','LineWidth',2),title('Critical Damping - Step'),grid
      xlabel('Time (sec)'),ylabel('System Response')
      subplot(222),plot(t,y2,'b','LineWidth',2),title('Over Damping - Step'),grid
      xlabel('Time (sec)'),ylabel('System Response')
      subplot(223),plot(t,y3,'g','LineWidth',2),title('Under Damping - Step'),grid
      xlabel('Time (sec)'),ylabel('System Response')
      subplot(224),plot(t,y4,'m','LineWidth',2),title('No Damping - Step'),grid
      xlabel('Time (sec)'),ylabel('System Response')
%
%   Model Conversion in Matlab
%
%   For the case with m = k = 1 (numerically), the system transfer function
%   for the simple mass-spring-dashpot system is given as
%         G(s)  =  1/[s^2 + cs + 1)]   =  z(s)/p(s)
%   for various values of c (viscous damping factor)
%
%   State-Space to Transfer Function in Matlab
      disp('  Transfer Function Form')
      sys1t = tf(sys1s),   sys2t = tf(sys2s)
      sys3t = tf(sys3s),   sys4t = tf(sys4s)
%
%   Transfer Function to Zero-Pole in MATLAB
      disp('  Zero-Pole-Gain Form')
      sys1z = zpk(sys1t),   sys2z = zpk(sys2t)
      sys3z = zpk(sys3t),   sys4z = zpk(sys4t)
%
%   Sometimes the Residue or Partial Fraction form is useful
%   (not really in this case, but let's demonstrate it anyway)
      disp('  Residue Form')
%   note that the tfdata command defines cell arrays
      [n1,d1] = tfdata(sys1t);   [n2,d2] = tfdata(sys2t);
      [n3,d3] = tfdata(sys3t);   [n4,d4] = tfdata(sys4t);
%   let's extract the cell arrays into normal vectors for use in residue
      n1 = n1{1}, d1 = d1{1},     n2 = n2{1}, d2 = d2{1},
      n3 = n3{1}, d3 = d3{1},     n4 = n4{1}, d4 = d4{1},
%   now let's use the residue command
      [r1,pr1,kr1] = residue(n1,d1),   [r2,pr2,kr2] = residue(n2,d2)
      [r3,pr3,kr3] = residue(n3,d3),   [r4,pr4,kr4] = residue(n4,d4)
%
%   Getting Frequency Domain Plots is also easy in Matlab
%
%   Bode plots
      nfig = nfig+1;  figure(nfig)
      w = logspace(-2,2,400);
      subplot(2,2,1),bode(sys1s,'r',w),title('Critical Damping'),grid     % ss form
      subplot(2,2,2),bode(sys2s,'b',w),title('Over Damping'),grid         % ss form
      subplot(2,2,3),bode(sys3t,'g',w),title('Under Damping'),grid        % tf form
      subplot(2,2,4),bode(sys4t,'m',w),title('No Damping'),grid           % tf form
      h = findobj(gcf,'LineStyle','-'); nh = length(h);
      for i = 1:nh; set(h(i),'LineWidth',2); end
```

```
%
%   Nichols plots (log-mag vs angle)
      nfig = nfig+1;  figure(nfig)
      subplot(2,2,1),nichols(sys1s,'r',w),grid,title('Critical Damping')
      subplot(2,2,2),nichols(sys2s,'b',w),grid,title('Over Damping')
      subplot(2,2,3),nichols(sys3s,'g',w),grid,title('Under Damping')
      subplot(2,2,4),nichols(sys4s,'m',w),grid,title('No Damping')
      h = findobj(gcf,'LineStyle','-'); nh = length(h);
      for i = 1:nh; set(h(i),'LineWidth',2); end
%
%   Nyquist plots (Im vs Re)
      nfig = nfig+1;  figure(nfig)
      subplot(2,2,1),nyquist(sys1s,'r',w),grid,title('Critical Damping')
      subplot(2,2,2),nyquist(sys2s,'b',w),grid,title('Over Damping')
      subplot(2,2,3),nyquist(sys3s,'g',w),grid,title('Under Damping')
      subplot(2,2,4),nyquist(sys4s,'m',w),grid,title('No Damping')
      h = findobj(gcf,'LineStyle','-'); nh = length(h);
      for i = 1:nh; set(h(i),'LineWidth',2); end
%
      diary off                % turn off diary file
%
%   Note: Before leaving this demo, briefly illustrate the LTIVIEW command!!!
%   (use File/Import to bring in the systems of interest -- then explore...)
%   (note also that all the above frequency domain plots are "interactive")
      ltiview
%
%   end of demo




%
%   MODELDEMO.M   Sequence of Matlab sample programs that demonstrate
%               various modeling and simulation techniques (both time and
%               frequency domain) including a simple Simulink sample
%
%   Mass-Spring-Dashpot Mechanical System
%     my''(t) + cy'(t) + ky(t)  =  u(t)
%
%   This example has been broken into three parts (3 files), as follows:
%    Demo 1: simple simulation and model conversion techniques
%    Demo 2: modeling systems in block diagram form (the OLD way) (*** this file ***)
%    Demo 3: modeling with Simulink (the NEW way)
%
%   File prepared by J. R. White, UMass-Lowell (last update: Feb. 2020)
%


      clear all,   close all
      format compact
%
%   open diary file for saving solutions
      delete modeldemo2.out,    diary modeldemo2.out
      disp('  *** MODELDEMO2.OUT  ***   Diary File for MODELDEMO2.M '),   disp(' ')
%
%   Modeling Systems Given in Block Diagram Form (The OLD Way)
%
%   Define the constants for the Under Damping case
      m = 1,   k = 1,   c = 1.2
%
%   Step 1 -- Setup individual transfer functions (see notes for block diagram)
      n1 = 1/m,   d1 = [1 c/m],   sys1 = tf(n1,d1);
      n2 = 1,     d2 = [1 0],     sys2 = tf(n2,d2);
      n3 = k/m,   d3 = [1 c/m],   sys3 = tf(n3,d3);
%
%   Step 2 -- Build a block diagonal (unconnected) system
      sysuct = append(sys1,sys2,sys3);    sysucs = ss(sysuct)
%
%   Step 3 -- Specify interconnections among blocks
      q = [1 0 0;2 1 -3;3 2 0]
%
%   Step 4 -- Specify inputs and outputs
      iu = [1],   iy = [2]
%
%   Step 5 -- Connect system together (this is the desired state-space form)
```

```
      syscs = connect(sysucs,q,iu,iy)
%
      syscsm = minreal(syscs)  % find a minimal realization of SS system
      syscz = zpk(syscsm)      % convert to zero, pole, gain form
      sysct = tf(syscz)        % convert to transfer function form
%
      diary off                % turn off diary file
%
%   end of demo



%
%   MODELDEMO.M   Sequence of Matlab sample programs that demonstrate
%               various modeling and simulation techniques (both time and
%               frequency domain) including a simple Simulink sample
%
%   Mass-Spring-Dashpot Mechanical System
%     my''(t) + cy'(t) + ky(t)  =  u(t)
%
%   This example has been broken into three parts (3 files), as follows:
%    Demo 1: simple simulation and model conversion techniques
%    Demo 2: modeling systems in block diagram form (the OLD way)
%    Demo 3: modeling with Simulink (the NEW way)  (*** this file ***)
%
%   File prepared by J. R. White, UMass-Lowell (last update: Feb. 2020)
%


      clear all,   close all
      format compact
%
%   open diary file for saving solutions
      delete modeldemo3.out,    diary modeldemo3.out
      disp('  *** MODELDEMO3.OUT  ***   Diary File for MODELDEMO3.M '),   disp(' ')
%
%   Modeling Systems Given in Block Diagram Form with Simulink (The NEW Way)
%
%   Define the constants for the Under Damping case
      m = 1,   k = 1,   c = 1.2
%
%   Display Simulink graphical model on screen
      modeldemo3_sl
%
%   Extract linear model from SIMULINK graphical model (saved in modeldemo3_sl.slx)
      disp('  Data from graphical SIMULINK model')
      [A,B,C,D] = linmod('modeldemo3_sl'),   syss = ss(A,B,C,D);
      sysz1 = zpk(syss)        % convert to zero, pole, gain form
      syssm = minreal(syss)    % find a minimal realization of SS system
      sysz2 = zpk(syssm)       % convert to ZPK form (after min. realization)
%
      diary off                % turn off diary file
%
%   end of demo
```

Table 5.2 Listing of Matlab diary files modeldemo1/2/3.out.

```
  *** MODELDEMO1.OUT  ***   Diary File for MODELDEMO1.M

m =
     1
k =
     1
c =
    2.0000    2.5000    1.2000         0
  State Space Models
A1 =
     0     1
    -1    -2
A2 =
         0    1.0000
   -1.0000   -2.5000
A3 =
         0    1.0000
   -1.0000   -1.2000
A4 =
     0     1
    -1     0
B =
     0
     1
C =
     1     0
D =
     0
  Transfer Function Form

Transfer function:
      1
-------------
s^2 + 2 s + 1


Transfer function:
       1
---------------
s^2 + 2.5 s + 1


Transfer function:
       1
---------------
s^2 + 1.2 s + 1


Transfer function:
   1
-------
s^2 + 1

  Zero-Pole-Gain Form

Zero/pole/gain:
   1
-------
(s+1)^2


Zero/pole/gain:
      1
-------------
(s+2) (s+0.5)


Zero/pole/gain:
        1
-----------------
(s^2  + 1.2s + 1)
```

```
Zero/pole/gain:
    1
----------
(s^2  + 1)

  Residue Form
n1 =
         0         0    1
d1 =
     1     2     1
n2 =
     0     0     1
d2 =
    1.0000    2.5000    1.0000
n3 =
         0         0    1
d3 =
    1.0000    1.2000    1.0000
n4 =
         0         0    1
d4 =
     1     0     1
r1 =
         0
    1.0000
pr1 =
    -1
    -1
kr1 =
     []
r2 =
   -0.6667
    0.6667
pr2 =
   -2.0000
   -0.5000
kr2 =
     []
r3 =
        0 - 0.6250i
        0 + 0.6250i
pr3 =
  -0.6000 + 0.8000i
  -0.6000 - 0.8000i
kr3 =
     []
r4 =
        0 - 0.5000i
        0 + 0.5000i
pr4 =
        0 + 1.0000i
        0 - 1.0000i
kr4 =
     []
```

***** MODELDEMO2.OUT *** Diary File for MODELDEMO2.M**

```
m =
     1
k =
     1
c =
    1.2000
n1 =
     1
d1 =
    1.0000    1.2000
n2 =
     1
d2 =
     1     0
```

```
n3 =
     1
d3 =
    1.0000    1.2000

a =
          x1    x2    x3
   x1  -1.2     0     0
   x2     0    -0     0
   x3     0     0  -1.2

b =
       u1  u2  u3
   x1   1   0   0
   x2   0   1   0
   x3   0   0   1

c =
       x1  x2  x3
   y1   1   0   0
   y2   0   1   0
   y3   0   0   1

d =
       u1  u2  u3
   y1   0   0   0
   y2   0   0   0
   y3   0   0   0

Continuous-time model.
q =
     1     0     0
     2     1    -3
     3     2     0
iu =
     1
iy =
     2

a =
          x1    x2    x3
   x1  -1.2     0     0
   x2     1     0    -1
   x3     0     1  -1.2

b =
       u1
   x1   1
   x2   0
   x3   0

c =
       x1  x2  x3
   y1   0   1   0

d =
       u1
   y1   0

Continuous-time model.
1 state removed.

a =
              x1          x2
   x1  2.14e-031     -1.414
   x2     0.7071       -1.2

b =
              u1
   x1  2.14e-031
   x2    -0.7071

c =
               x1           x2
   y1           1  8.863e-032
```

```
d =
       u1
   y1   0

Continuous-time model.

Zero/pole/gain:
        1
-----------------
(s^2  + 1.2s + 1)


Transfer function:
       1
---------------
s^2 + 1.2 s + 1
```

***** MODELDEMO3.OUT *** Diary File for MODELDEMO3.M**

```
m =
     1
k =
     1
c =
    1.2000
  Data from graphical SIMULINK model
A =
         0    1.0000   -1.0000
         0   -1.2000         0
    1.0000         0   -1.2000
B =
     0
     1
     0
C =
     1     0     0
D =
     0

Zero/pole/gain:
         (s+1.2)
-------------------------
(s+1.2) (s^2  + 1.2s + 1)

1 state removed.

a =
            x1       x2
   x1  -0.2464  -0.4607
   x2    1.661  -0.9536

b =
         u1
   x1   0.5
   x2  -0.5

c =
           x1      x2
   y1  0.7071  0.7071

d =
       u1
   y1   0

Continuous-time model.

Zero/pole/gain:
        1
-----------------
(s^2  + 1.2s + 1)
```