Matlab IV: Modeling and Simulation

Updated: August 2012

Table of Contents

Note: See Matlab I: Getting Started for more information about this tutorial series including its organization and for more information about the Matlab software. Before proceeding with this tutorial, download Part_4_Modeling_and_Simulation.zip . This zip file contains the example files needed for this portion of the tutorial.

Section 9: Modeling and Simulation

Matlab has several auxiliary Toolboxes distributed by MathWorks, Inc. which are useful in constructing models and simulating dynamical systems. These include the System Identification Toolbox, the Optimization Toolbox, and the Control System Toolbox. These toolboxes are collections of m-files that have been developed for specialized applications. There is also a specialized application, Simulink, which is useful in modular construction and real time simulation of dynamical systems.

System Identification

The System Identification Toolbox contains many features for processing experimental data and is used for testing the appropriateness of various models by optimizing values of model parameters. It is particularly useful in working with dynamical systems data and time series analyses. This toolbox is included in the Matlab installations on the ITS time sharing servers. The identification process is a bit complex, but a guided tour through a simple example can be accessed with the *iddemo* command at a Command Window prompt.

Using the Control System Toolbox

The Control System Toolbox contains routines for the design, manipulation and optimization of LTI (linear time invariant) systems of the form

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

where matrices A, B ,C, and D time invariant are constants. These routines can be used individually or as post-processing tools for a system created with Simulink. The Control System Toolbox also supports two auxiliary applications, the LTI Viewer and the SISO Design Tool. The LTI Viewer is basically used to plot graphs of the system response due to various inputs and the SISO Design Tool is used to design single input-single output systems, i.e., systems for which the input and output vectors have dimensions 1 by 1. Information about the LTI Viewer is

available from *Help > Product Help > Control System Toolbox > Getting Started > Analyzing Models > LTI Viewer* and a viewer window can be launched from the Command window with

```
>> ltiview
```

We don't comment on the SISO design tool since that would require knowledge of control theory, but we give an example of the use of LTI Viewer. Consider the system

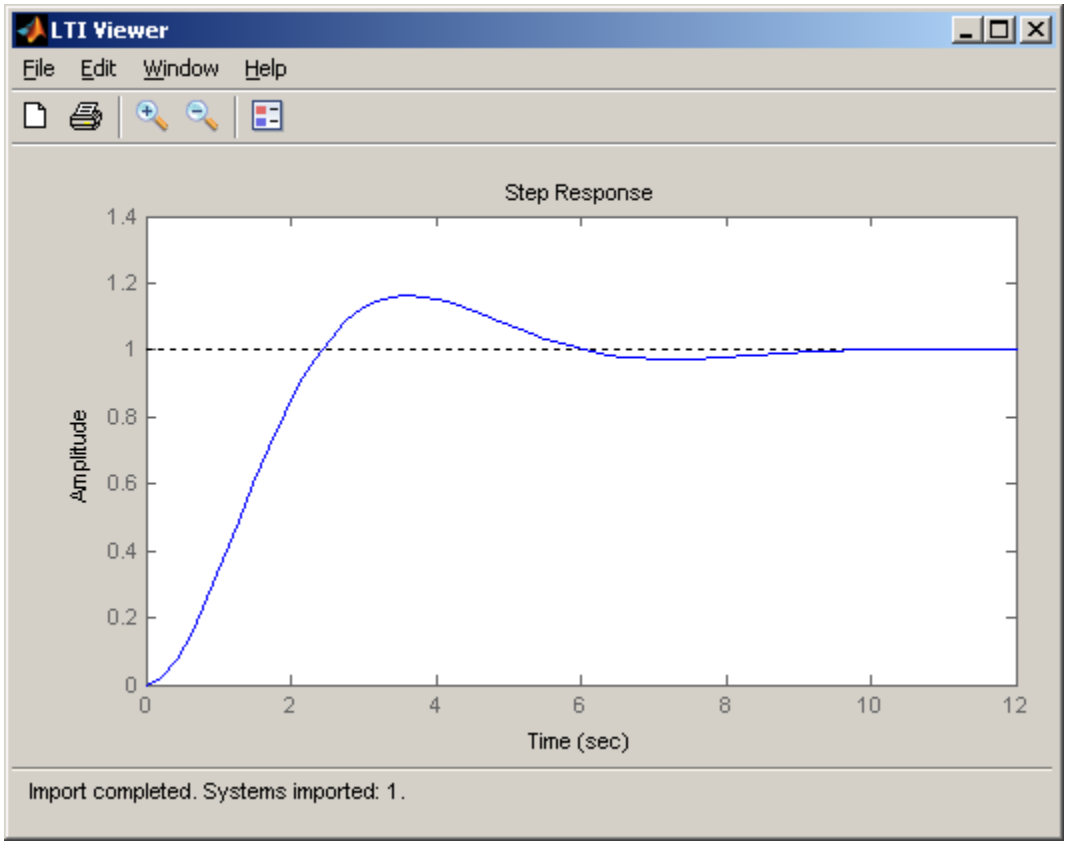
$$A = \begin{bmatrix} 0 & 1 \\ -1 & 1 \end{bmatrix}, B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, C = [0 \ \ 1], D = [0]$$

To import the system to the LTI Viewer, we create a system object using the *ss* command which generates a state space (ss).

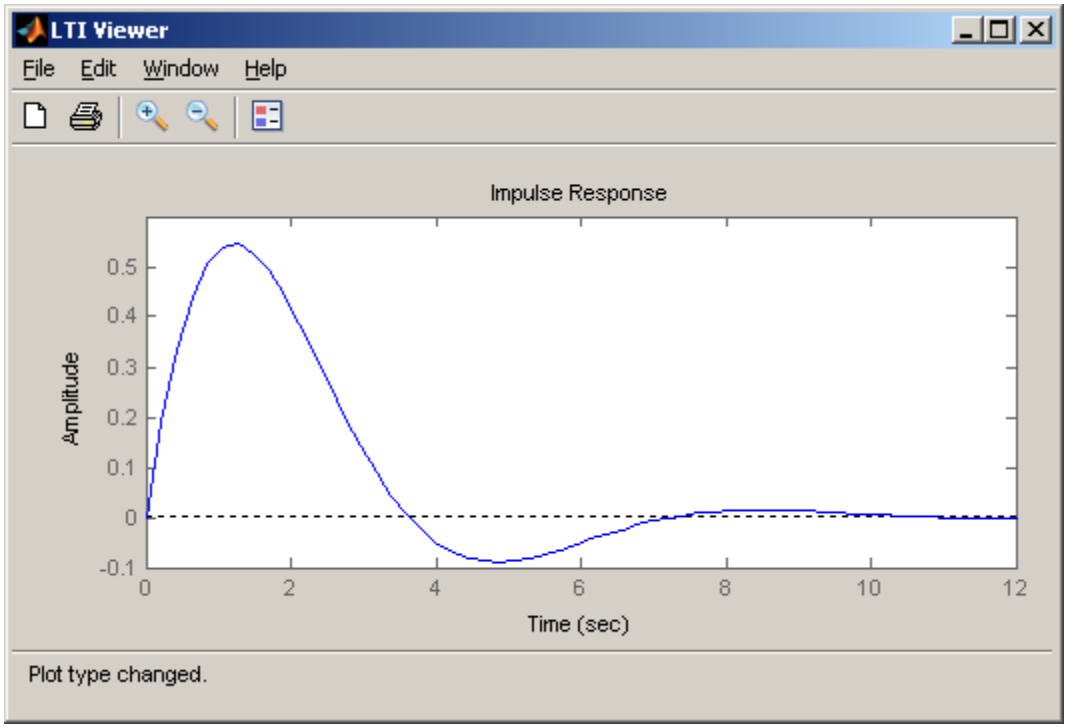
```
>> A=[0 1;-1 -1];
>> B=[0 1]';
>> C=[1 0];
>> D=0;
>> s1=ss(A,B,C,D)

a =
       x1  x2
   x1   0   1
   x2  -1  -1


b =
       u1
   x1   0
   x2   1


c =
       x1  x2
   y1   1   0


d =
       u1
   y1   0
```

```
Continuous-time model.
```

Then from *File* > *Import* on the LTI Viewer window select *s1* and click on the *OK* button. The *s1* system will then appear in the viewer.

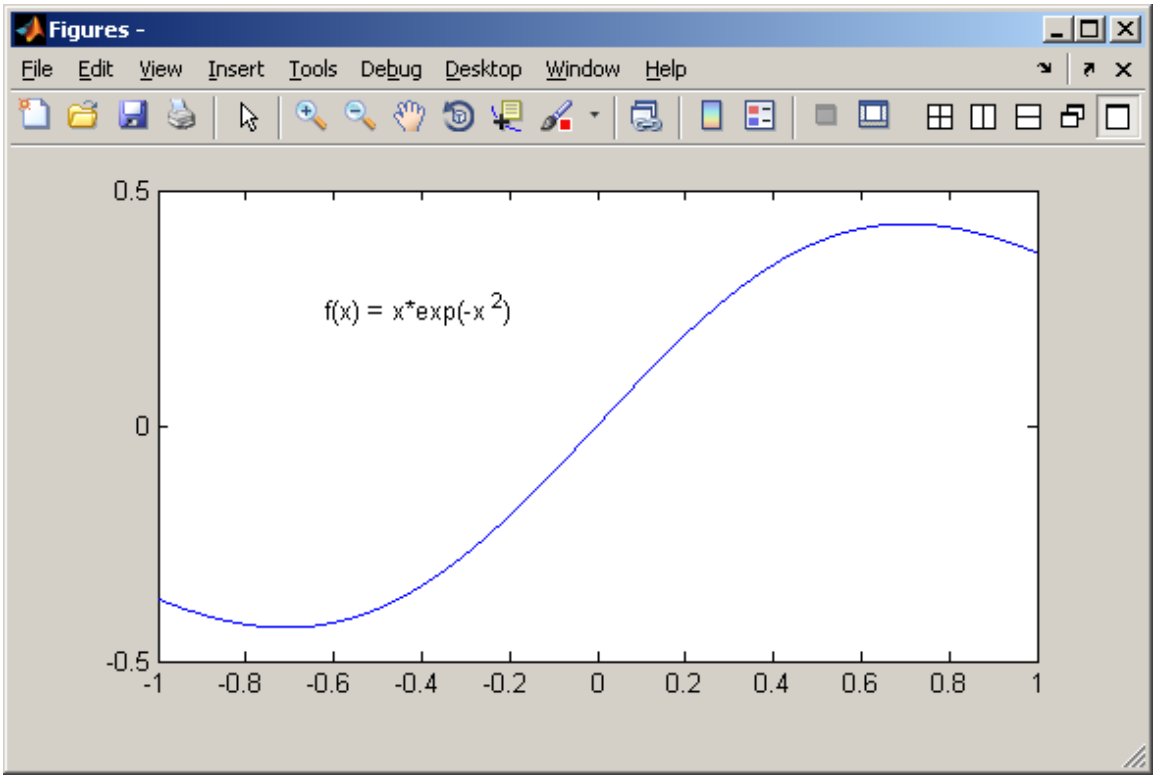


Next, right click on that figure and select *Plot Types* > *Impulse* to get the following figure, which is a plot of the response of the system to a unit impulse at time zero.



Optimization Toolbox

The Optimization Toolbox offers a rich variety of routines used for the minimization and maximization of functions under constraints. We will describe only two simple and commonly used examples. The first one is **fminbnd**, which determines the location within a given interval of its argument values that a function attains its minimum functional value. Note that the maximum of a function $f(x)$ is equal to the negative of the minimum of $-f(x)$; hence we can use **fminbnd** to compute locations of maxima of functions also. Suppose now that we want to compute the minimum and maximum values of $f(x) = x \cdot e^{-x^2}$ in the interval $[-1,1]$ for the argument *x*.



From visual inspection of the graphic we can see that there is a minimum functional value for an argument near -0.7 and maximum functional value for an argument near +0.7. Furthermore for this simple case we can determine from calculus that the first derivative is zero for arguments of $x = \pm\frac{1}{\sqrt{2}}$, such that $f(x) = \pm\frac{1}{\sqrt{2}}e^{-\frac{1}{2}}$. Thus we know the correct analytic answer, but we can also determine the solution numerically with Matlab's **fminbnd** optimization function.

At the command line we type:

```
>> x = fminbnd('x*exp(-x^2)',-1,1)

x =

   -0.7071

>> x*exp(-x^2)

ans =

   -0.4289
```

```
>> x=fminbnd('-x*exp(-x^2)',-1,1)

x =

    0.7071


>> -(-x*exp(-x^2))

ans =

     0.4289
```

Thus, the minimum and maximum values of $f(x) = x \cdot e^{-x^2}$ are -0.4289 and 0.4289 and are attained when *x* is equal to `-0.7071` and `0.7071` respectively.

Next, consider the problem of linear optimization, which is frequently encountered in operations research and mathematical economics. The objective is to find *n* real numbers that minimize the linear expression

$$c_1 x_1 + c_2 x_2 + ... + c_n x_n$$

subject to the constraints:

$$a_{11} x_1 + a_{12} x_2 + ... + a_{1n} x_n \le b_1$$
$$a_{21} x_1 + a_{22} x_2 + ... + a_{2n} x_n \le b_2$$
$$\dots$$
$$a_{m1} x_1 + a_{m2} x_2 + ... + a_{mn} x_n \le b_m$$

or, in matrix form, minimize $c^{\mathrm{T}} x$ such that $Ax \le b$.

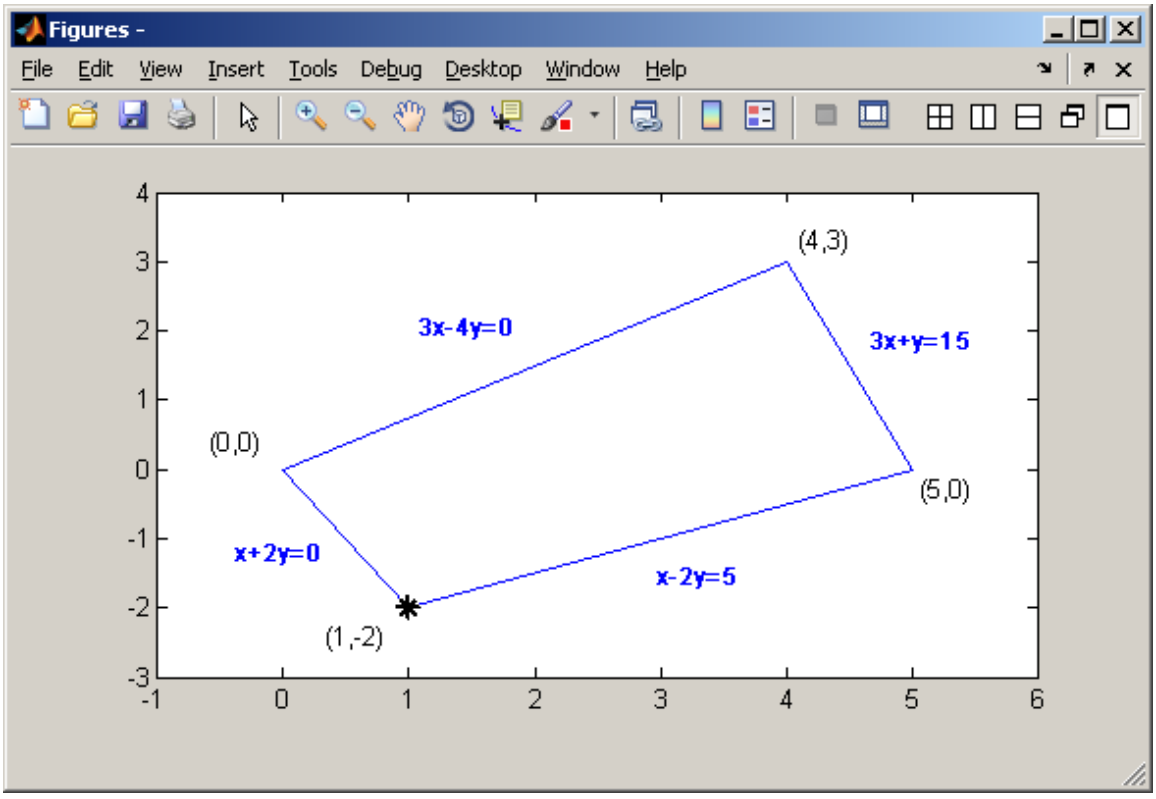
The problem can be solved via the function *linprog*. As an example consider the following 2-D linear optimization problem:

Minimize $2x_1 + 3x_2$, so that the constraints
$-3x_1 + 4x_2 \le 0$, $3x_1 + x_2 \le 15$, $x_1 - 2x_2 \le 5$, $-2x_1 - x_2 \le 0$ hold.

To solve it we type:

```
>> c=[2 3];
>> A=[-3 4; 3 1; 1 -2; -2 -1];
>> b=[0 15 5 0]';
>> linprog(c,A,b)
>>ans =

     1.0000
    -2.0000
```

Next we give a geometrical interpretation of this solution. It can be shown that the pair (x1,x2) that solves the problem, is one of the vertices of the quadrilateral in the following figure:



The edges of the trapezoid correspond to the optimization constraints and the points in its interior satisfy all of them, hence the solution must be attained in the trapezoid or on its boundary. Due to a theorem in linear optimization, the solution is attained at one of the trapezoids' vertices, in this case at the point (1,-2).

Using Simulink

Simulink is a simulation tools library for use in modeling dynamical systems in modular block form. Most deterministic transformative systems in nature can roughly be thought of as a "black

box" receiving an input vector of information *u* and eliciting a unique output vector of information *y*. In the case that both *u* and *y* vary with time we are talking about dynamic systems.

Associated with a system is the so-called state vector which loosely speaking contains the required information at time t_0 that together with knowledge of the input for time greater than t_0, uniquely determines the output for $t \geq t_0$. A general continuous dynamical system can be modeled by using the following set of ordinary differential and algebraic equations:

$$\frac{dx}{dt} = f(t, x, u)$$
$$y = g(t, x, u)$$

for $t \geq t_0$ and $x(t_0) = x_0$, where f, g are general (possibly non-linear functions). In the following we will consider only linear systems of the form:

$$\frac{dx}{dt} = Ax + Bu \qquad (1)$$
$$y = Cx + Du$$

where A, B, C, D are matrices and x, u, y the state, input and output vectors respectively.
Of course the dimensions of *A*, *B*, *C*, and *D* are such that the matrix manipulations on the right hand side of (1) are well defined.

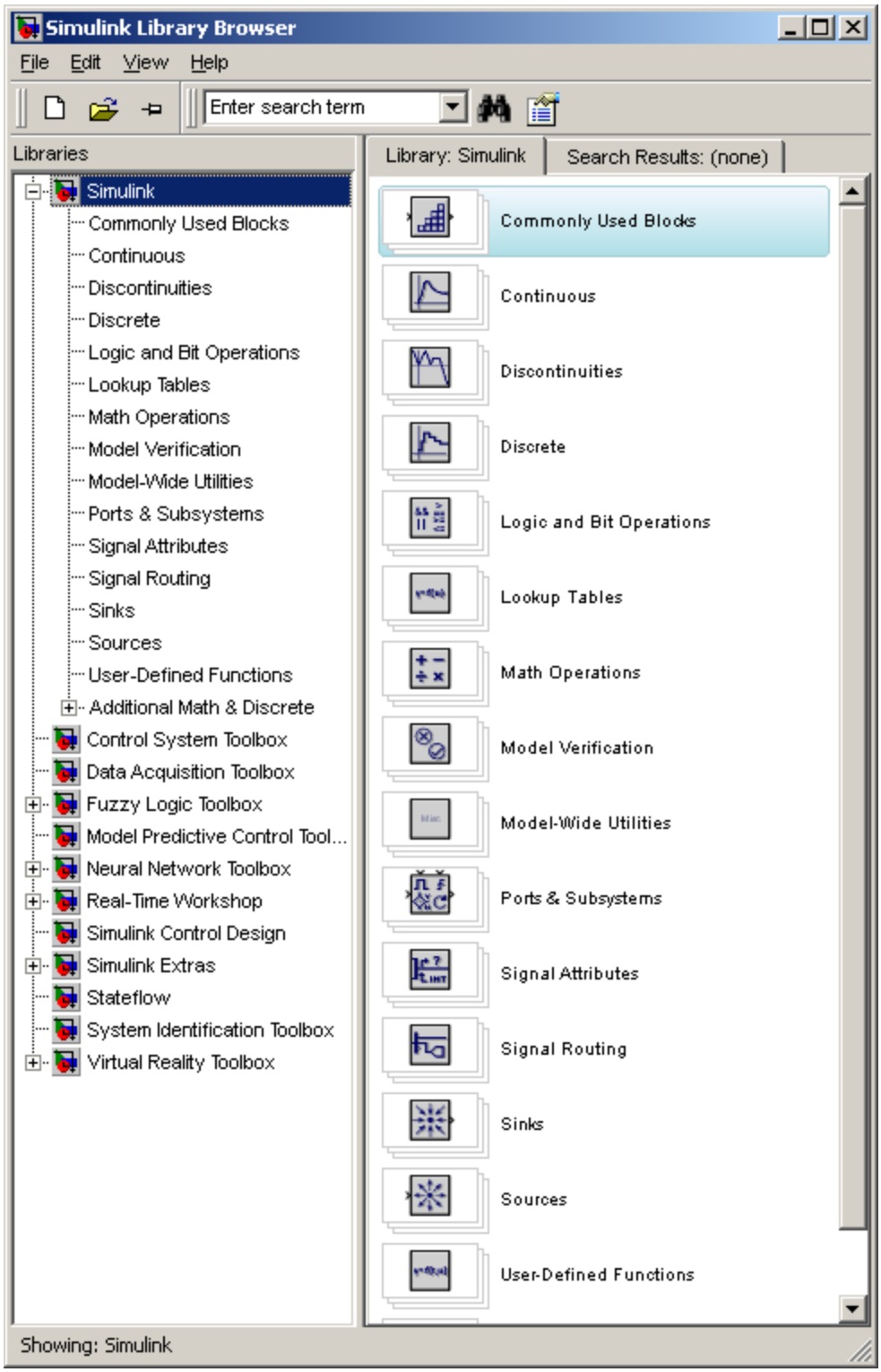
Simulink Library Browser

Simulink can be launched by typing

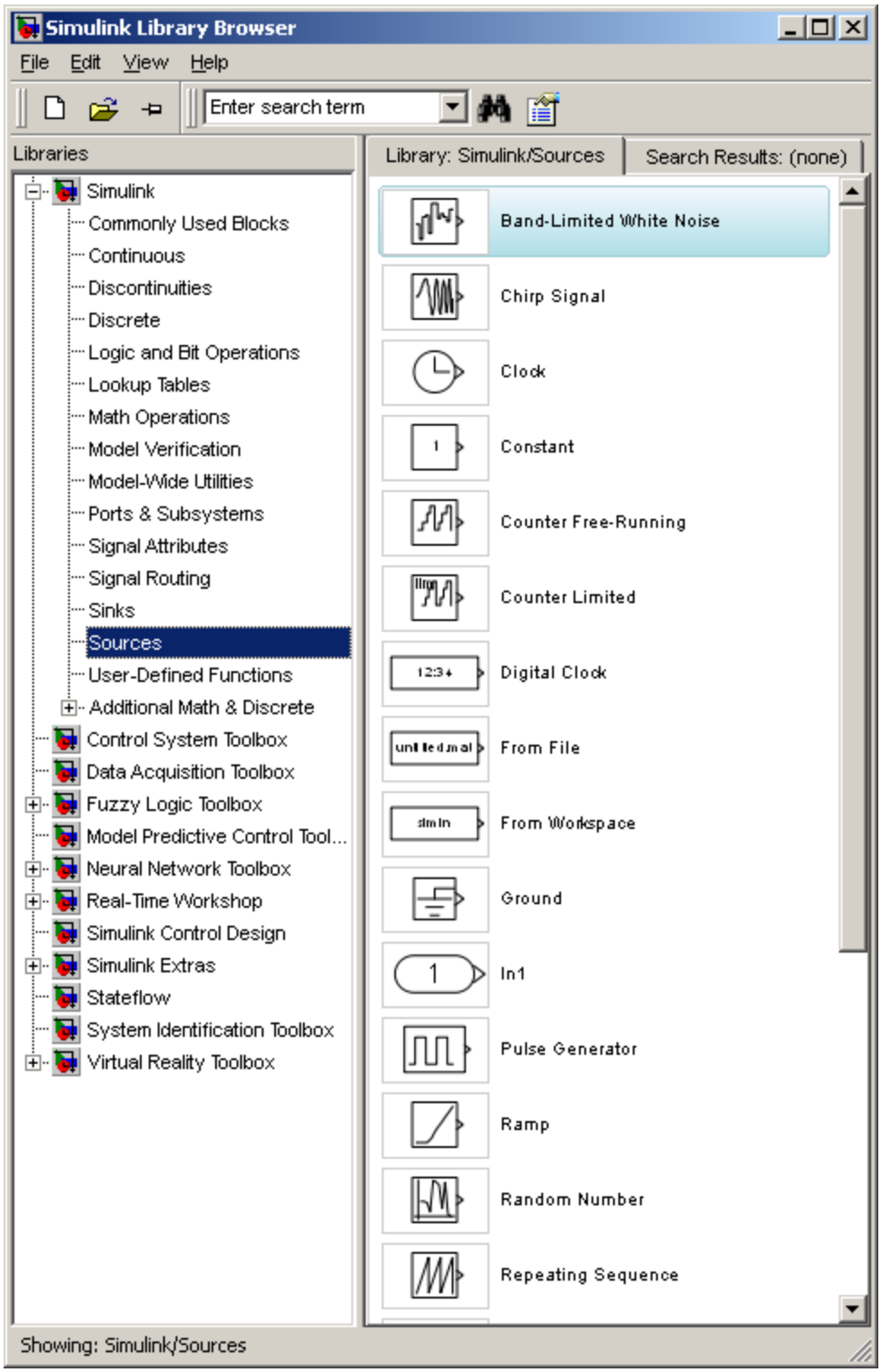
```
>> simulink
```

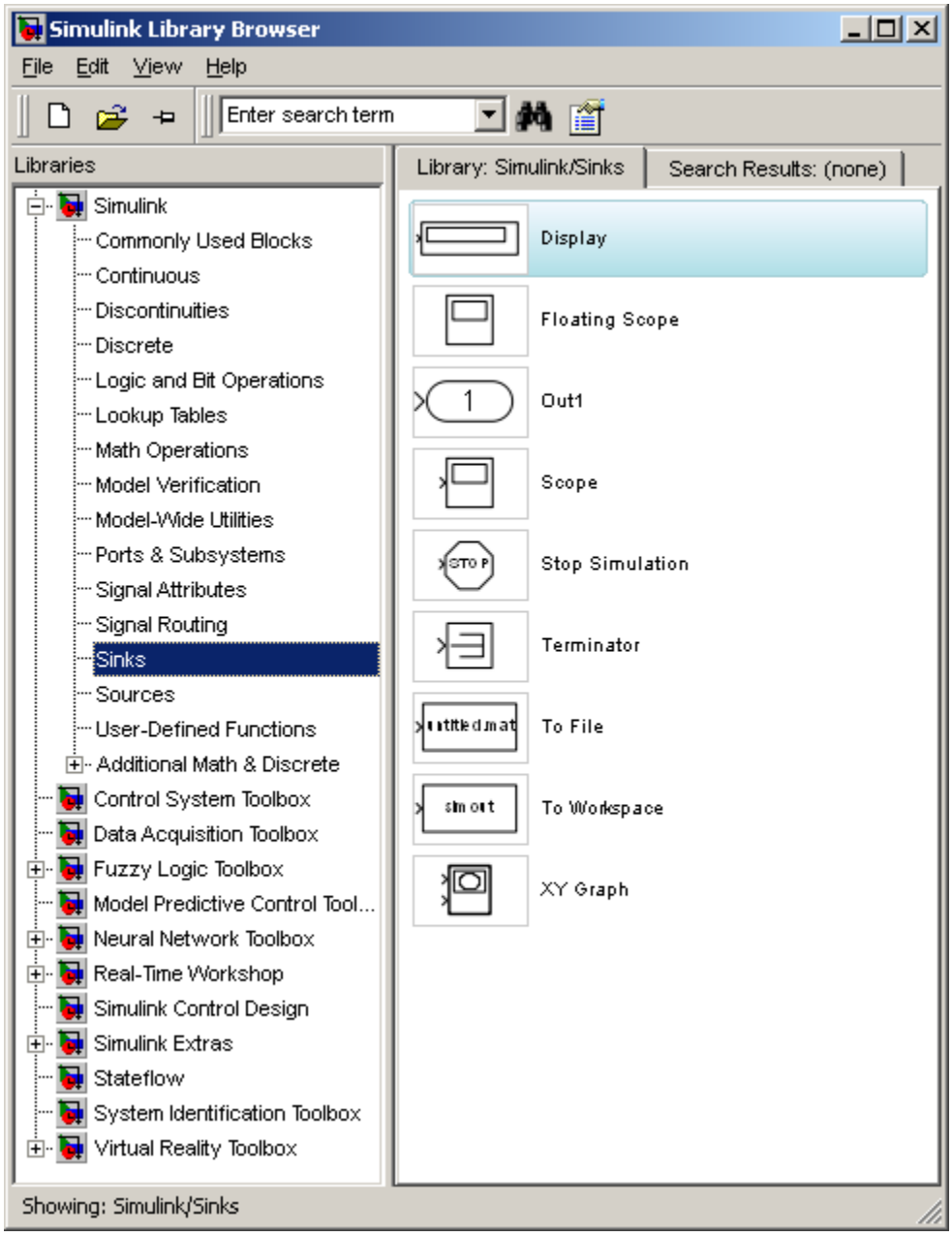
in the Command window of the default Matlab desktop, by clicking on the Simulink icon on Matlab's icon toolbar  , or from the *Start* button with *Start* > *Simulink* > *Library Browser*.

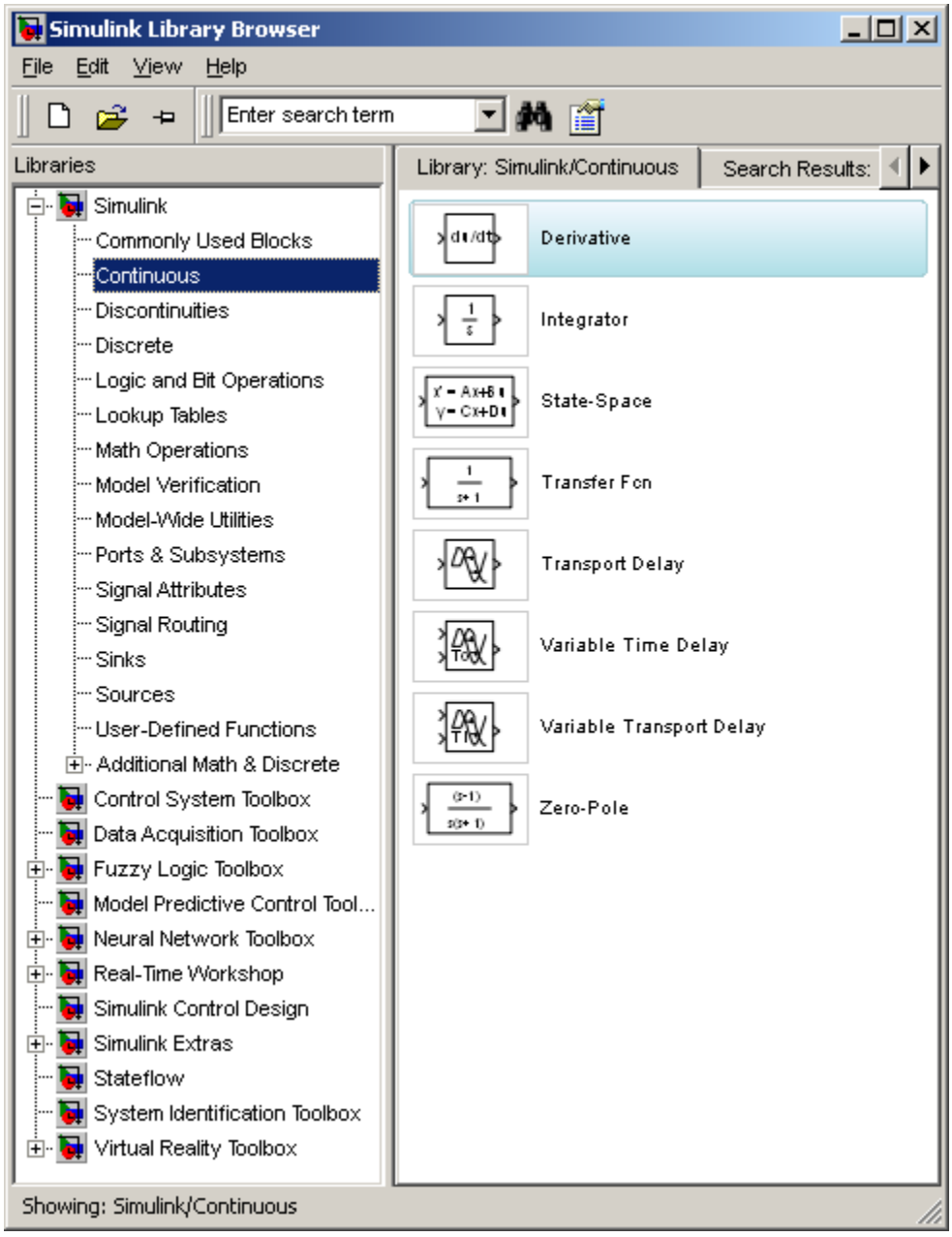
A Simulink library browser is brought up in a floating window. In the left panel there are various categories of libraries, some of which can be expanded or collapsed. Clicking on any item in that panel will fill a right side panel with icons for simulation blocks of that particular category. These block icons have appearances similar to standard icons used in engineering diagrams, and can be imported into a window where a model is being constructed. The model elements all need to be connected to the system as a whole in one way or another, so each block element will have at least one input or output position, but can also have multiple positions of both types. Each block has a default name in the library, with which it is initially labeled when exported. However, once inserted into a diagram the default name of the block can be altered to be something more descriptive of what will be its actual function.


Simulink Library Browser
File Edit View Help
Enter search term
Libraries
Simulink
Commonly Used Blocks
Continuous
Discontinuities
Discrete
Logic and Bit Operations
Lookup Tables
Math Operations
Model Verification
Model-Wide Utilities
Ports & Subsystems
Signal Attributes
Signal Routing
Sinks
Sources
User-Defined Functions
Additional Math & Discrete
Control System Toolbox
Data Acquisition Toolbox
Fuzzy Logic Toolbox
Model Predictive Control Tool...
Neural Network Toolbox
Real-Time Workshop
Simulink Control Design
Simulink Extras
Stateflow
System Identification Toolbox
Virtual Reality Toolbox
Library: Simulink
Search Results: (none)
Commonly Used Blocks
Continuous
Discontinuities
Discrete
Logic and Bit Operations
Lookup Tables
Math Operations
Model Verification
Model-Wide Utilities
Ports & Subsystems
Signal Attributes
Signal Routing
Sinks
Sources
User-Defined Functions
Showing: Simulink

The library's functionalities are divided into several groups (click on any of the category icons for both the UNIX or Windows versions). For example the categories *Sources* and *Sinks* contain various kinds of inputs and ways to handle or display the output respectively. Also, it contains the group *Continuous* that will be used later when dealing with dynamical systems.

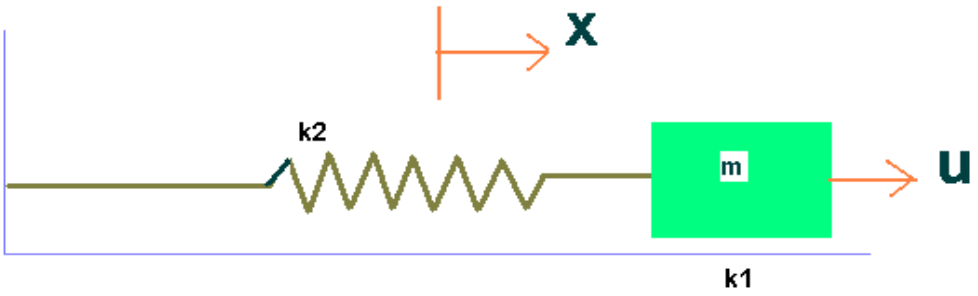

Simulink Library Browser
File Edit View Help
Enter search term
Libraries
Simulink
Commonly Used Blocks
Continuous
Discontinuities
Discrete
Logic and Bit Operations
Lookup Tables
Math Operations
Model Verification
Model-Wide Utilities
Ports & Subsystems
Signal Attributes
Signal Routing
Sinks
Sources
User-Defined Functions
Additional Math & Discrete
Control System Toolbox
Data Acquisition Toolbox
Fuzzy Logic Toolbox
Model Predictive Control Tool...
Neural Network Toolbox
Real-Time Workshop
Simulink Control Design
Simulink Extras
Stateflow
System Identification Toolbox
Virtual Reality Toolbox
Library: Simulink/Sources
Search Results: (none)
Band-Limited White Noise
Chirp Signal
Clock
Constant
Counter Free-Running
Counter Limited
Digital Clock
From File
From Workspace
Ground
In1
Pulse Generator
Ramp
Random Number
Repeating Sequence
Showing: Simulink/Sources


Simulink Library Browser
File Edit View Help
Enter search term
Libraries
Simulink
Commonly Used Blocks
Continuous
Discontinuities
Discrete
Logic and Bit Operations
Lookup Tables
Math Operations
Model Verification
Model-Wide Utilities
Ports & Subsystems
Signal Attributes
Signal Routing
Sinks
Sources
User-Defined Functions
Additional Math & Discrete
Control System Toolbox
Data Acquisition Toolbox
Fuzzy Logic Toolbox
Model Predictive Control Tool...
Neural Network Toolbox
Real-Time Workshop
Simulink Control Design
Simulink Extras
Stateflow
System Identification Toolbox
Virtual Reality Toolbox
Library: Simulink/Sinks
Search Results: (none)
Display
Floating Scope
1
Out1
Scope
STOP
Stop Simulation
Terminator
untitled.mat
To File
simout
To Workspace
XY Graph
Showing: Simulink/Sinks



Construction/ Simulation of Dynamical Systems

In the following, we consider a simple physical example to illustrate the usage of Simulink. One of the simplest systems introduced in mechanics classes is the vibrating spring,



which can be modeled by the ordinary differential equation

$$my'' + k_1 y' + k_2 y = u \qquad (2)$$

Here m is the mass of the body supported by the spring; k_1 and k_2 are the viscous and spring friction coefficients respectively, and u is the force applied to the body. The unknown function y is the distance of the body from the equilibrium position. Our first observation is that the differential equation describing the motion of the body is of second order (in other words the highest differentiation order of the equation is 2). To reduce it to a system of differential equations of first order (so that we can use Simulink) we make the following substitution:

$$x_1 = y$$
$$x_2 = y'$$

Then (1) becomes:

$$\frac{dx_1}{dt} = x_2$$
$$\frac{dx_2}{dt} = -\frac{k_2}{m}x_1 - \frac{k_1}{m}x_2 + \frac{1}{m}u$$

or in matrix form:

$$\begin{bmatrix} dx_1/dt \\ dx_2/dt \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k_2}{m} & -\frac{k_1}{m} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} \cdot u \qquad (3)$$

Here x_1 and x_2 are the state variables and u the input to the system. The output can be selected in various ways depending on what characteristics of the system are desired to be measured; it

could be x_1 (that is displacement), x_2 (velocity) or a linear combination of x_1, x_2, and u. For our purposes we simply define the output to be x_1. That is,

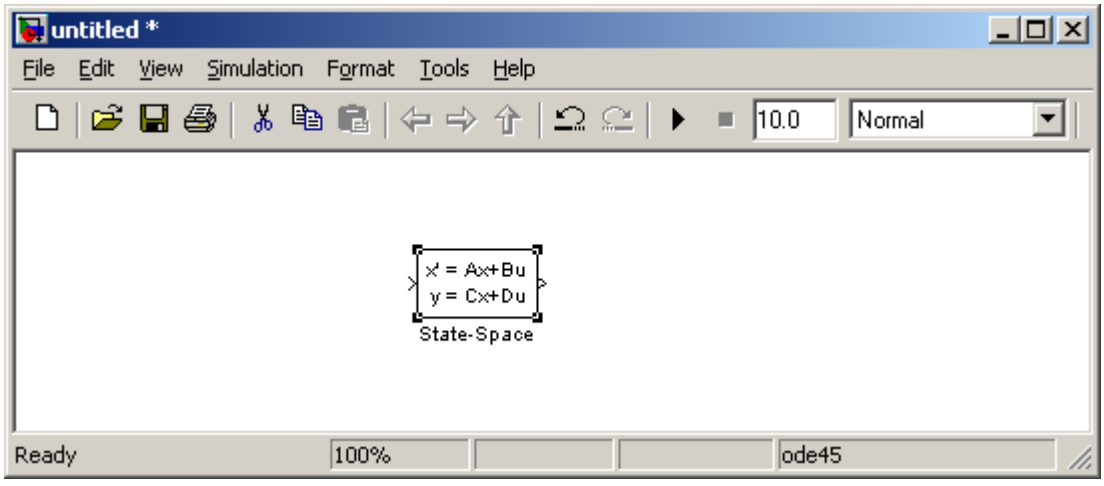
$$y = [1 \quad 0]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + [0]u \qquad (4)$$

in matrix form.

Equations (3) and (4) constitute the representation of the system in the form (1), with

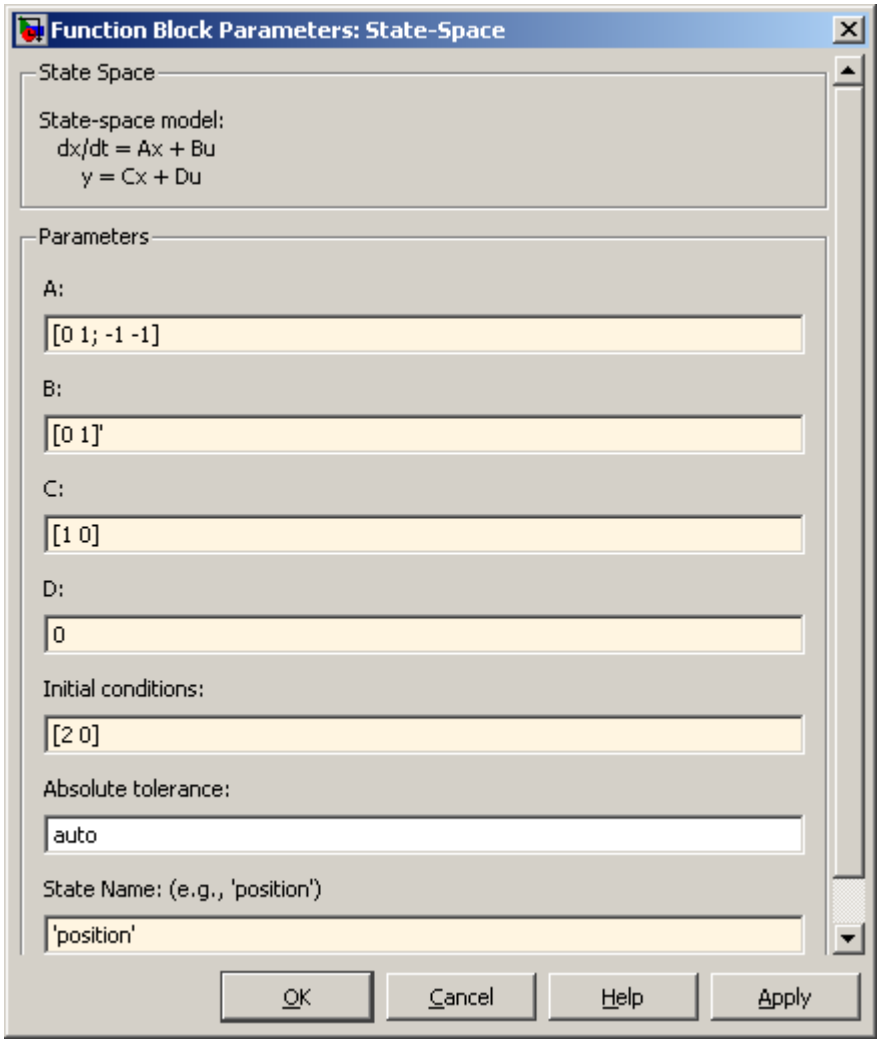
$$A = \begin{bmatrix} 0 & 1 \\ -\frac{k_2}{m} & -\frac{k_1}{m} \end{bmatrix}, B = \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix}, C = [1 \quad 0], D = [0]$$

Furthermore, we take $m = k_1 = k_2 = 1$, that is $A = \begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. Our initial conditions are $x_1(0) = 2$, $x_2(0) = 0$; that is at time $t = 0$ the body is located at a distance of 2 units from the equilibrium position and its velocity is 0. The next step is to build the system using Simulink. Clicking on the *Create a new model* button on the upper left corner of the Simulink library browser launches an untitled new model window. Next double click on the *Continuous* button in the *Simulink Library Browser* window, select the *State-Space* icon and drag it into the new model window.



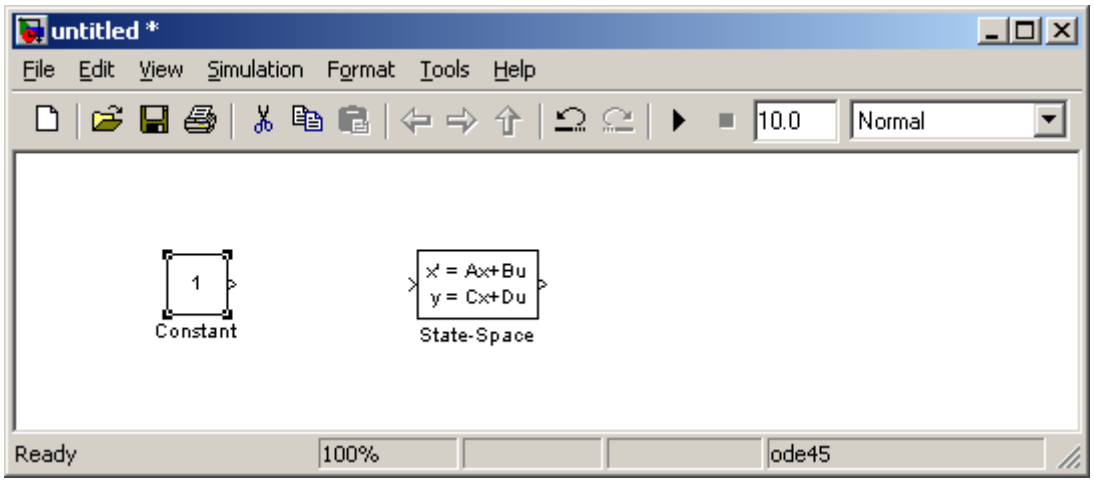
After double clicking on the dragged *State-Space* icon, the *Function Block Parameters* window appears in which we specify the matrices A, B, C and D as well as the initial condition vector. Note that the matrices are entered in the one row format described in section 3, but for large

matrices it is more convenient to define A, B, C, D in the command line (possibly with other names) and simply enter their names in the block parameters window. We can also label the state variable as 'position':

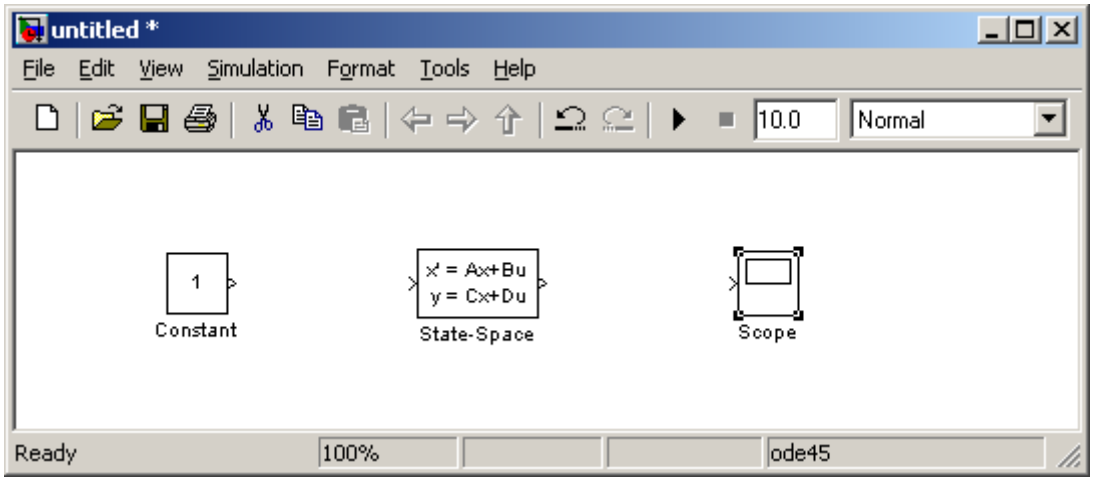


To assure that the changes are saved, click on the *Apply* button and then the *OK* button.

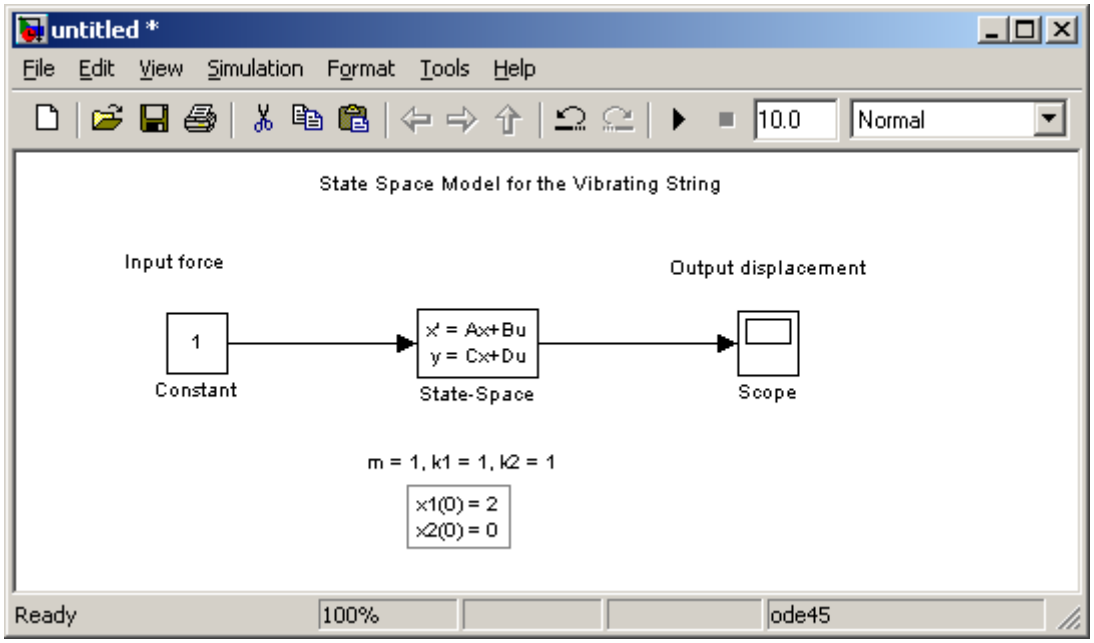
In the following we double click the *Sources* icon to select an input for the system. Let's choose the input *Constant* and drag it into the new model window. Again, by double clicking on the dragged *Constant* icon we specify the value of the constant input. For our example we take $u = 1$.



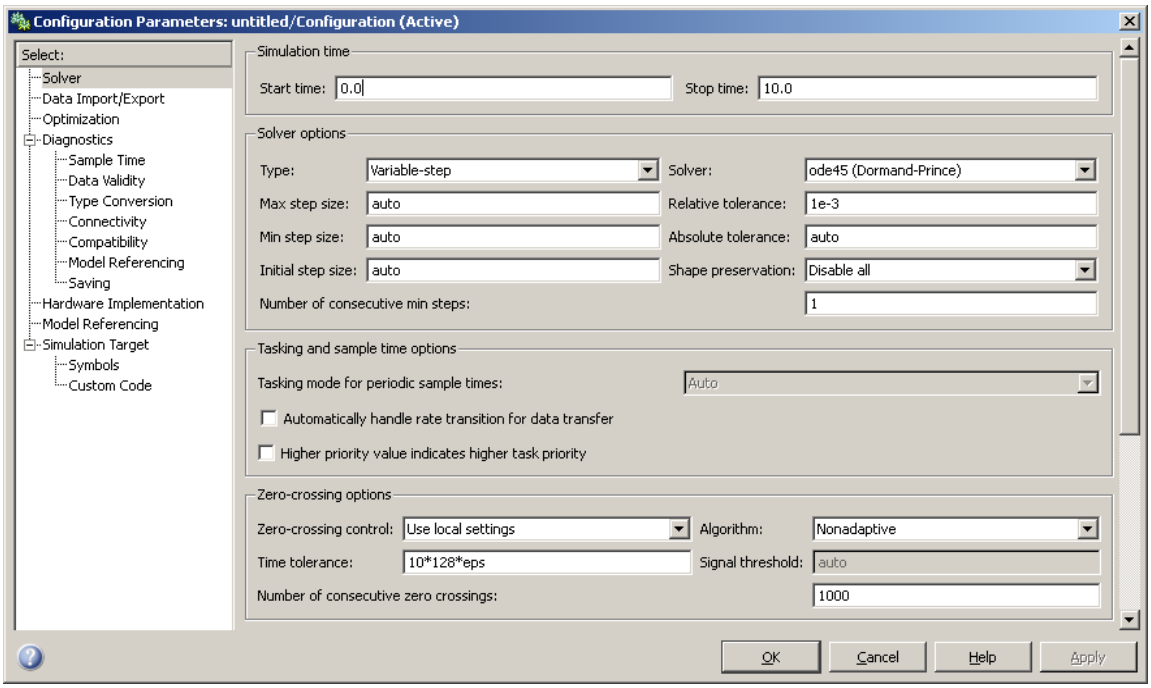
The output is selected by clicking on the *Sinks* button. Choose for example the *Scope* icon and drag it into the model window (Scope provides a graph of the system's output).



Next, connect the system blocks with arrows. For example, to connect the constant and state-space blocks, click on the right arrow of the constant block and move the cursor to the left arrow of the state-space block while holding the left mouse key down. Another method is to select the arrow origin block then click on the arrow destination block while holding down the control key. We can also add text on the model window by double clicking at any point of it and inserting the desired information.

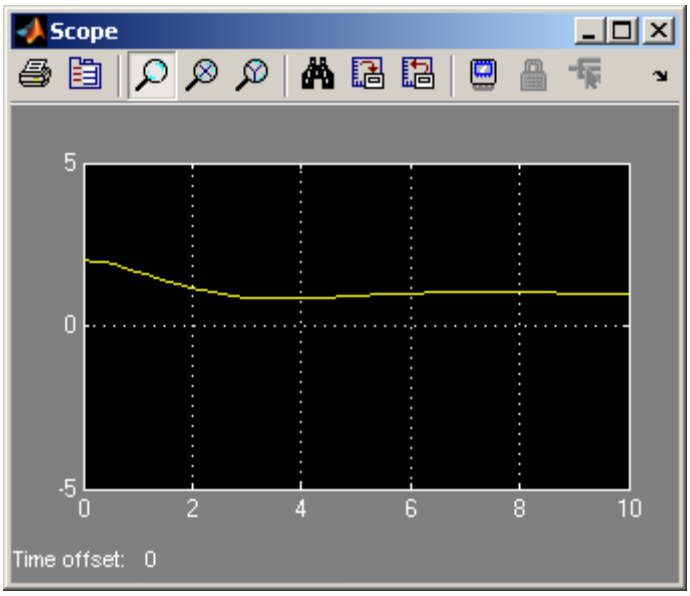


Then, from the model window toolbar, select *Simulation* > *Configuration Parameters* to specify the simulation parameters (the simulation initial and final time and the ODE solver to be used for example). In this example we choose $t_{final} = 10$.

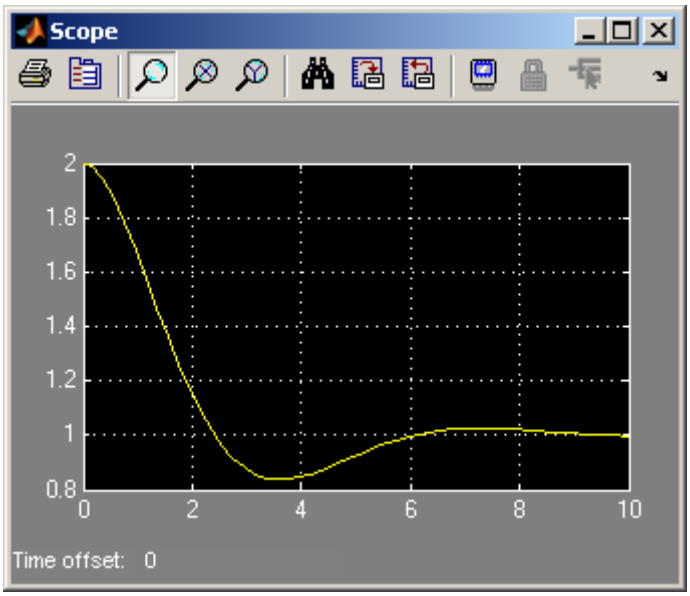


Now that the system has been built up, we are ready to run the simulation, in other words numerically solve the system of ODEs to obtain the output y. Simply, press the *Start simulation*

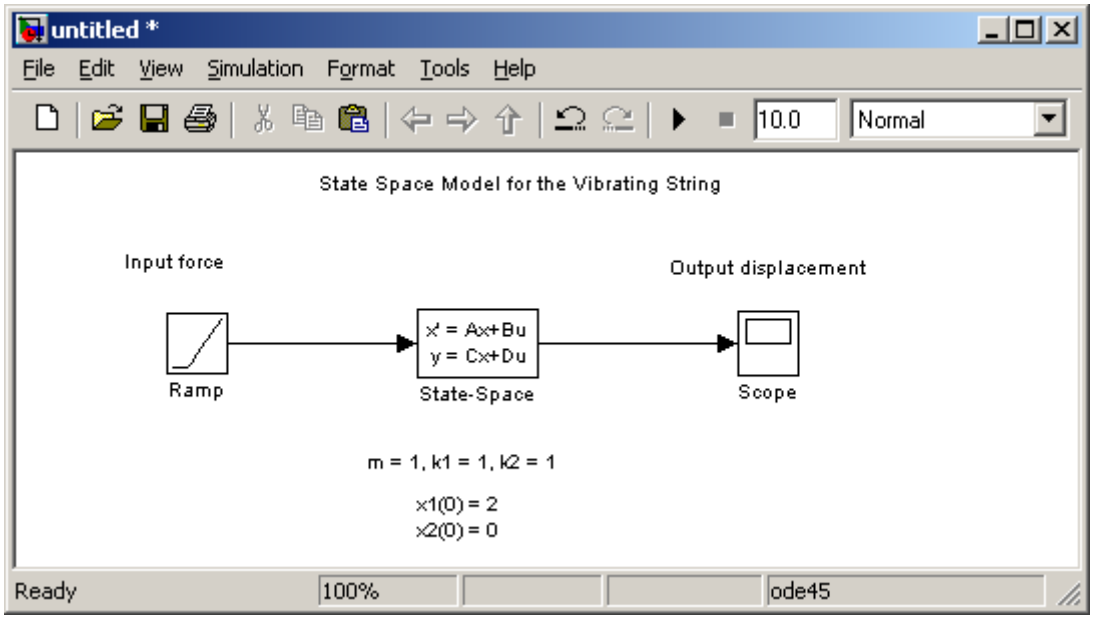
button on the *Simulation* toolbar menu (►) and then double click on the *Scope* icon to obtain a plot of the output.

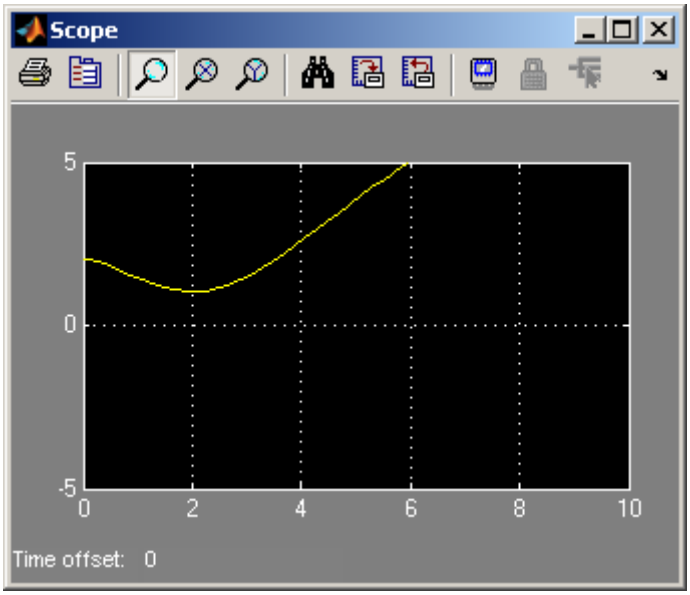


To get automatic scaling in the plot, click on the binoculars icon in the Scope's toolbar.



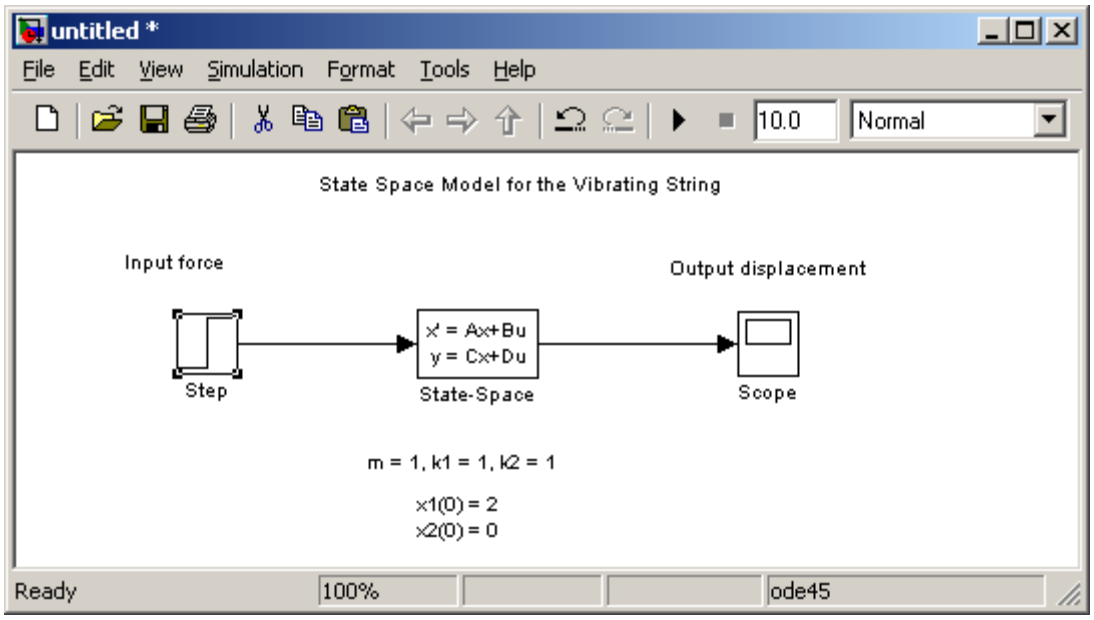
We observe that after time t is approximately equal to 9, the displacement asymptotically approaches a constant value. Next we change the system's input to a ramp by following the previous procedure. We specify the ramp input parameters by double-clicking on the Ramp icon from Sources and after double clicking on that icon choosing slope=1, start time=0 and initial output=0. In the following figures we present the modified system and its response.

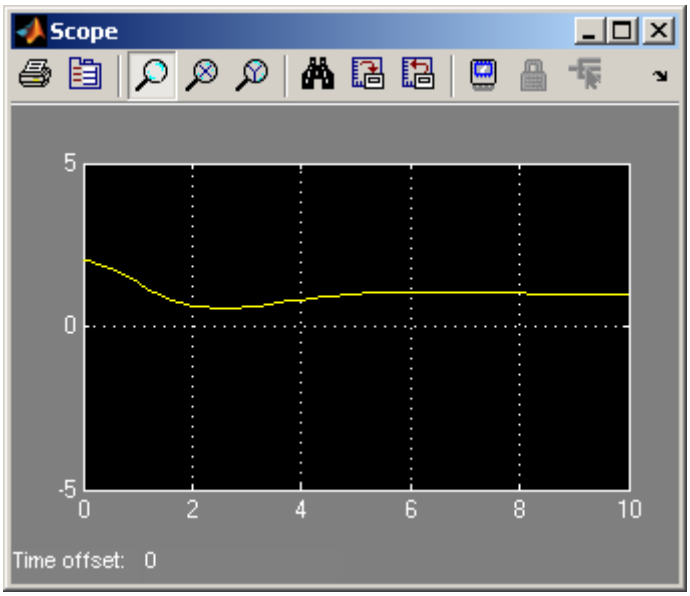




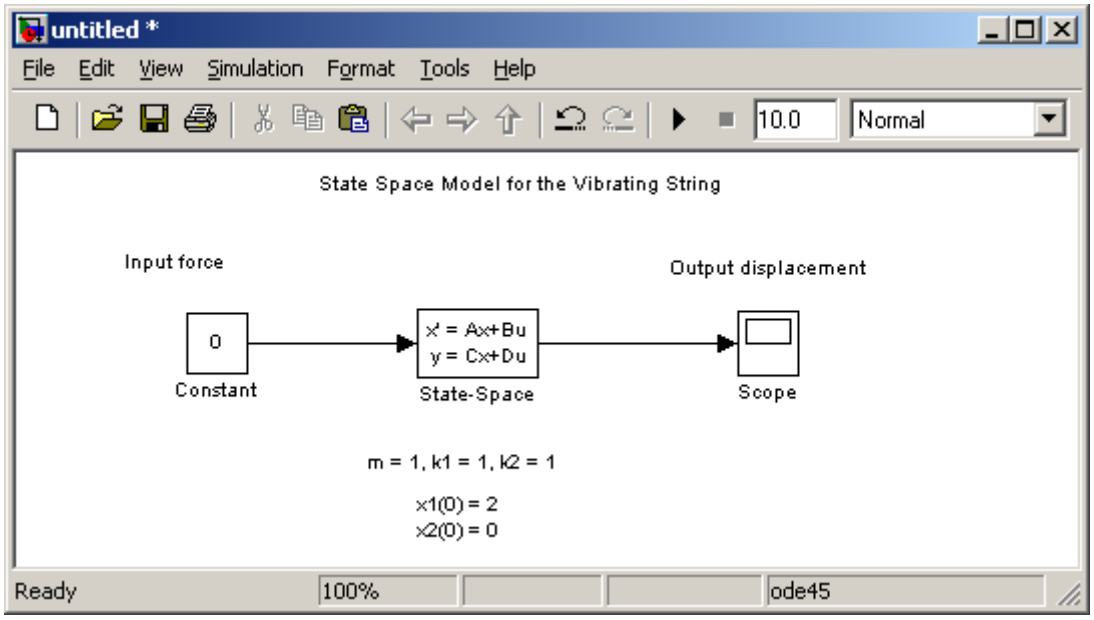
As expected, since the force acting on the body is increasing in magnitude and doesn't change direction, the distance from the rest point increases. Also, we consider the case of a step input, i.e., a force spontaneously changing its magnitude at some time t_0. Again the step function

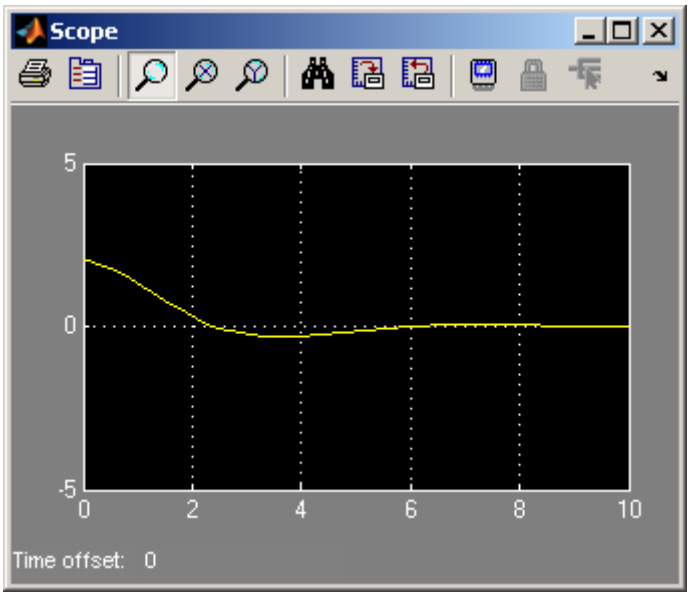
parameters, are specified by double-clicking on the *Step* icon after it is dragged in from *Sources* and then choosing parameters: *step time* = 1, *initial value* = 0, and *final value* = 1.



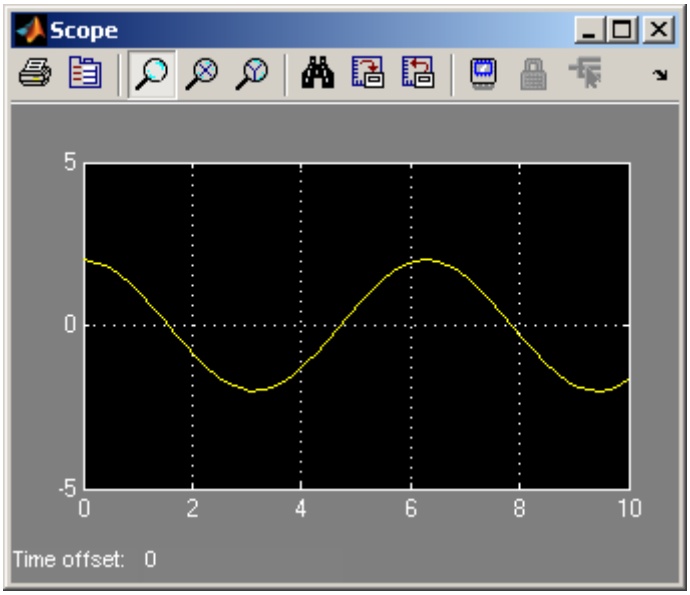


In the sequel, let's consider the case of zero force acting on the body, i.e., we have a constant input equal to zero. We expect that the body will eventually return to the rest position $x_1 = 0$ and indeed this is what our model predicts.





In all previous cases except the ramp input's case, the system's output eventually approaches a constant value. Let's examine the case where viscous friction is negligible, that is $k_1 = 0$ and again the input force is the constant 0 (double click on *State-Space* icon and adjust A to [0 1; -1 0]). Again click on the *Apply* button and the *OK* button. We expect that the body will be moving internally, since no force exists to counterbalance the spring's force acting on it. Indeed, the prediction of our model is consistent with the physical intuition. As we can see in the Scope window after running the simulation again the output of the system (displacement of the body) oscillates between the extreme values –2 and 2.



In our last example we try to correct this behavior and actually make the system's output approach the value 0 (that is to make the body return to the equilibrium position) by introducing a so-called feedback control law. More precisely we define a new input to the system which is not user supported as in the previous examples, but depends on the system's output. In other words, the system's output simultaneously defines its input. Indeed, we define:

$u = K \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$, where $K = [0 \; -1]$. The selection of the feedback gain matrix K is of course not arbitrary but we will not comment on its derivation. In fact K can be selected in many different ways and it is the task of the control engineer to determine the best one, depending on design requirements and limitations.

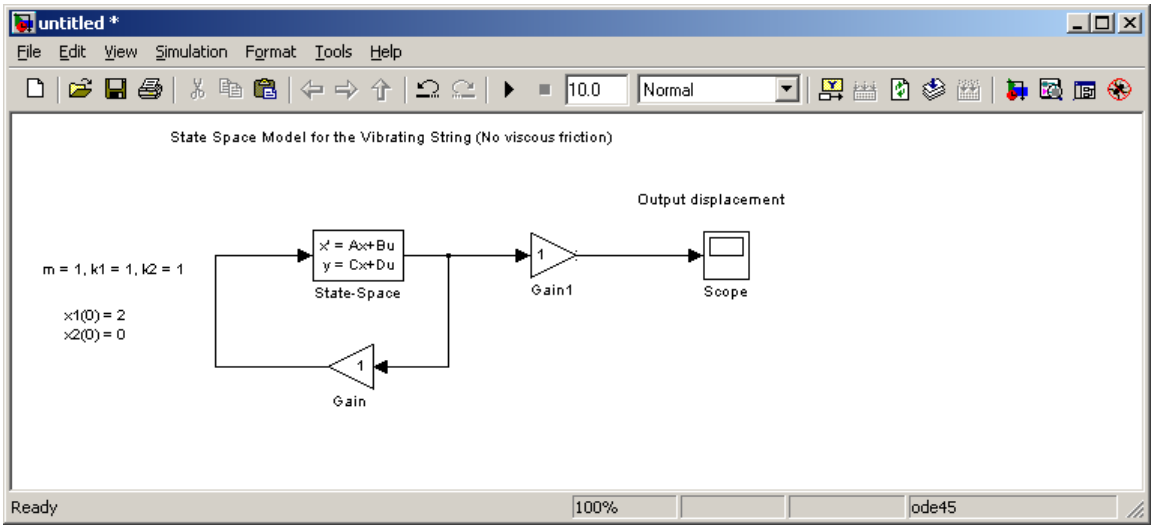
Then our system becomes:

$$\frac{dx}{dt} = (A + B \cdot K) \cdot x$$
$$y = [1 \quad 0] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + [0]u$$

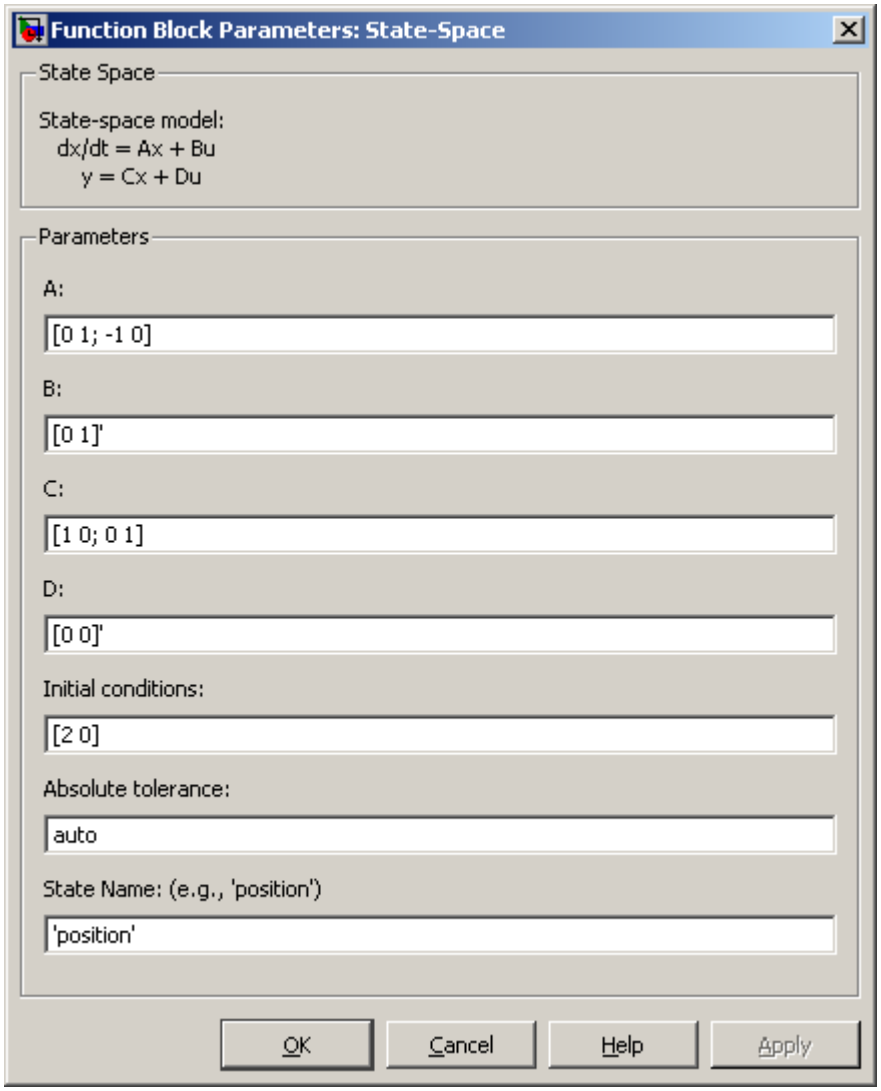
but using Simulink we actually develop the equivalent formulation

$$\frac{dx}{dt} = (A + B \cdot K) \cdot x$$
$$y_1 = x$$
$$y = [1 \quad 0] \cdot y_1$$

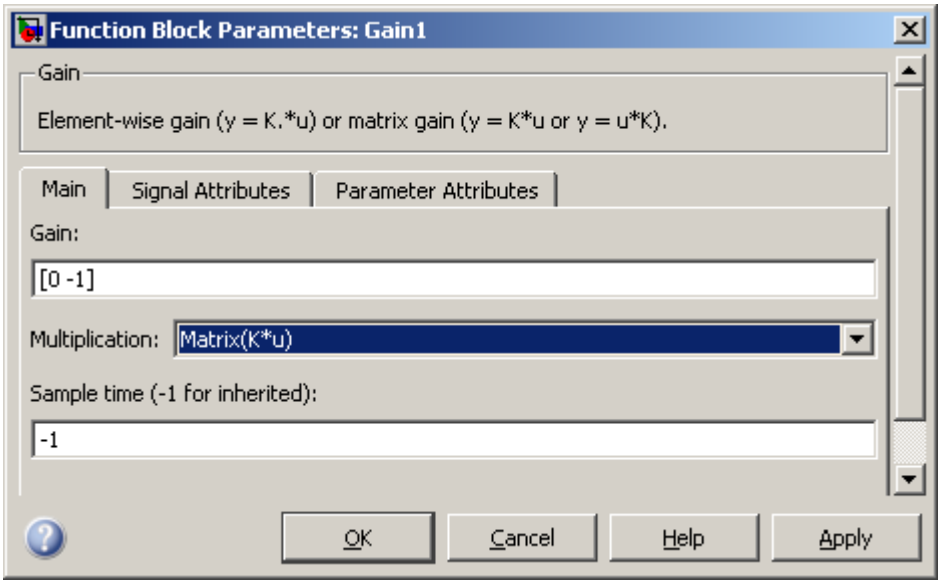
In the next figures the feedback controlled system is presented along with its output. To construct this model drag *Gain* blocks from the *Math Operations* in the Simulink library (and again double click on them to specify the gain matrices). You will see that if you drag two blocks into your model, the default name of the second one will be *Gain1*. Similarly, if your model required three gain blocks, the third one dragged in would have the default name *Gain2*, and so forth. Also, in order to rotate a *Gain* block for layout convenience simply right-click on it, choose the *Format* option from the menu that appears select the options *Rotate block* or *Flip block*. Finally, note that the body returns to its equilibrium position as it was desired.



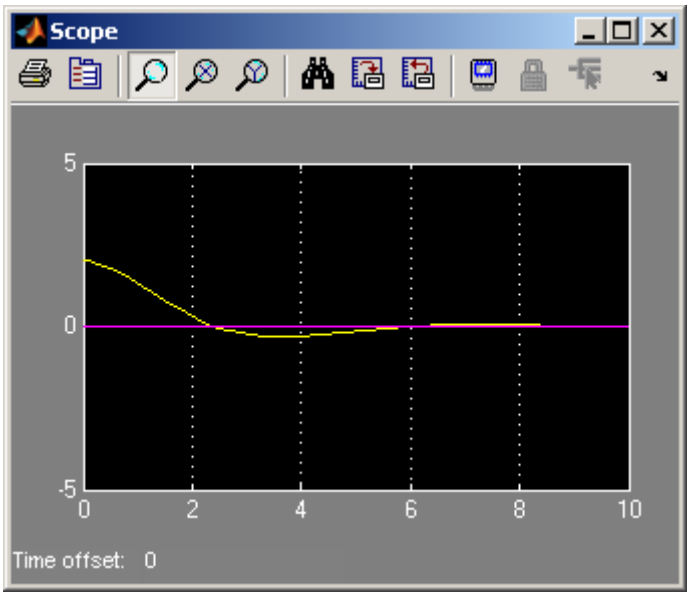
The output from the *State Space* block feeding into the *Gain* blocks now needs to be a two element vector and thus the *C* matrix needs expansion to [1 0; 0 1] and the *D* matrix needs expansion to [0 0]'.



Note that the *K* vector for the feedback gain is the one we used above, [0 -1]



whereas the gain feeding the scope will need to have a *K* vector [1 0]. Output to the scope will then be the just the displacement.



Simulink Subsystems

In Matlab programming, functions are used to encapsulate a computation so that it can be used repeatedly without having to duplicate code wherever it is needed. In addition, a function can insulate its calling script from having to worry about its implementation details. In Simulink, subsystems play a similar role. Using subsystems in Simulink has these advantages:

- It helps reduce the number of blocks displayed in the model window.
- Functionally related blocks can be kept together.
- It permits the establishment of a hierarchical block diagram, wherein a Subsystem block is on one layer and the blocks that make up that subsystem are on another.

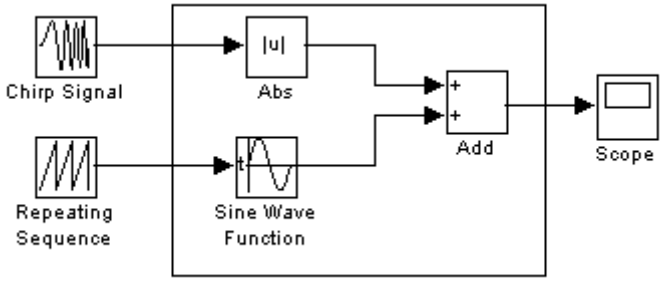
In Simulink, a subsystem can be created in two ways:

- One way is to add the blocks that make up the subsystem to the model, then group those blocks into a subsystem.
- The other way is to first add a Subsystem block to the model, then open that block and install the component blocks of the Subsystem to the subsystem window.

Grouping existing blocks into a Subsystem

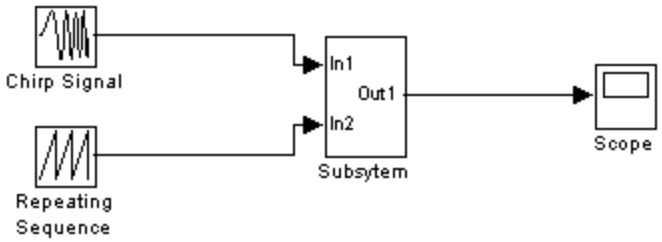
If a model already contains the blocks needed for a desired subsystem, you can create the subsystem by grouping those blocks:

1. Enclose the blocks and connecting lines that you want to include in the subsystem within a bounding box. For example, the figure below shows a model that does signal processing. The *Abs*, *Sine Wave Function* and *Add* blocks that do the signal conversions are selected within a bounding box. The box illustrated can be selected by clicking the mouse at the upper left position, and then while depressing the right mouse button drag to the lower right position



   The components within the box will be selected when the mouse button is released.

2. Choose *Create Subsystem* from the *Edit* menu. Simulink replaces the selected blocks with a Subsystem block. The figure below shows the model after the *Create Subsystem* command has been chosen. If necessary, the Subsystem block can be resized so that the port labels are readable).



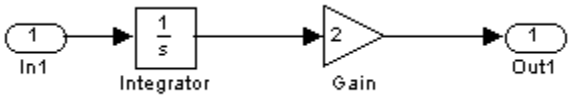
Creating a Subsystem by adding a Subsystem block

The process of creating a subsystem before adding its component blocks usually consists of three major steps:

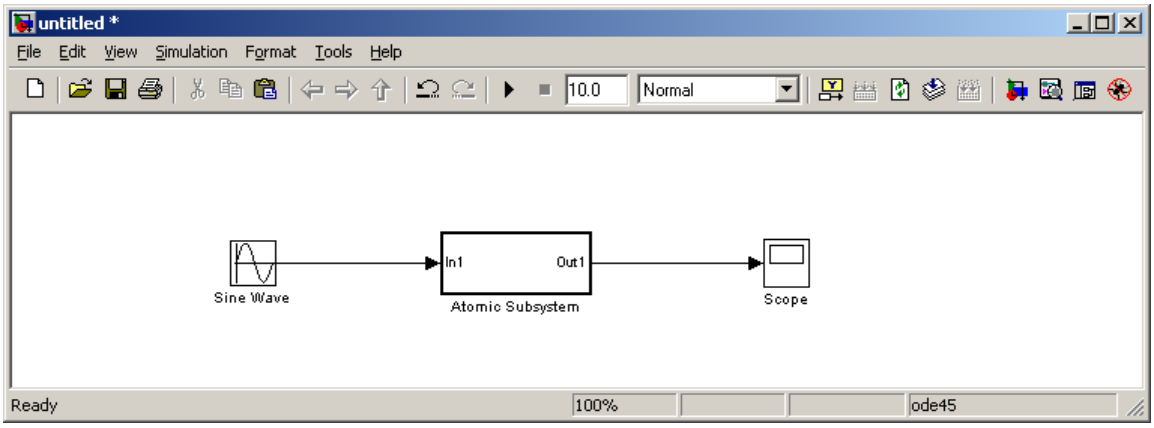
1 Copy the Subsystem block from the *Ports & Subsystems* library into your model.
2 Open the Subsystem block by double-clicking it. Simulink opens the subsystem in the current or a new model window, depending on the model window reuse mode that you selected.
3 In the empty Subsystem window, create the subsystem. Use *In* blocks to represent input from outside the subsystem and *Out* blocks to represent external output.

Here is an example which models an integrator system.

1. From the Simulink Library Browser, go to the Ports & Subsystems subdirectory. Click-drag the Atomic Subsystem block into the Model window.
2. Open the subsystem block by double-clicking the subsystem block.
3. Delete the line connecting In1 block and Out1 block.
4. Insert blocks into this subsystem window as prescribed in the figure below and then fix the block layout and connections between blocks.



5. Click on the *Go to parent system* button (up arrow on icon bar) or choose *View > Go To Parent* from the navigation bar. This will send you back to the underlying Model window.
6. Go back to the Simulink Library Browser and add sine wave function and scope blocks into the Model window as described below.



7. Run the simulation by clicking the *Start* button on the toolbar or by selecting *Simulation* > *Start* from the navigation bar. Results can be seen by double clicking on the scope block.

Simulink S-Functions

What Is An S-Function?

S-functions can be used to add your own custom blocks to a Simulink model. An S-function is a computer language description of a Simulink block written in Matlab, C, C++, Fortran, or Ada. The form of an S-function is very general and it can accommodate continuous, discrete, and hybrid systems.

How S-Functions Work

Execution of a Simulink model proceeds in stages. An S-function is comprised of a set of callback methods that perform tasks required at each simulation stage. During simulation of a model, at each simulation stage, Simulink calls the appropriate methods for each S-Function block in the model. Tasks performed by S-function methods include

- Initialization. In this stage, Simulink initializes the S-function. Specifically, this process includes initializing a simulation structure; setting the number and dimensions of input and output ports; setting the block sample times; and allocating storage areas and the *sizes* array.
- Calculation of next sample hit. This stage calculates the time of the next sample hit if a variable sample time block has been created,.
- Calculation of outputs in the major time step. After this call has been completed, all the output ports of the blocks are valid for the current time step.
- Update of discrete states in the major time step. In this call, all blocks should perform once-per-time-step procedures.
- Integration. This call is applicable to models with continuous states and/or non-sampled zero crossings. If an S-function has continuous states, Simulink calls its

output and derivative portions at minor time steps to compute the states for your S-function. If a MEX S-function written in C has non-sampled zero crossings, then Simulink calls its output and zero-crossings portions at minor time steps to locate the zero crossings.

Implementing S-Functions

An S-function can be implemented as either an m-file or a MEX file. Here we focus on m-file implementations. An m-file S-function consists of a Matlab function having the following form:

[*sys,x0,str,ts]=f(t,x,u,flag,p1,p2,...)*

where *f* is the S-function's name, *t* is the current time, *x* is the state vector of the corresponding S-function block, *u* represents the block's inputs, *flag* indicates a task to be performed, and *p1*, *p2*, ... are parameters defining the block. During simulation of a model, Simulink repeatedly invokes the S-function *f*, using *flag* value to indicate the task to be performed for a particular invocation. Each time the S-function performs the task, it returns the result in a structure having the [*sys,x0,str,ts]=f(t,x,u,flag,p1,p2,...)* format.

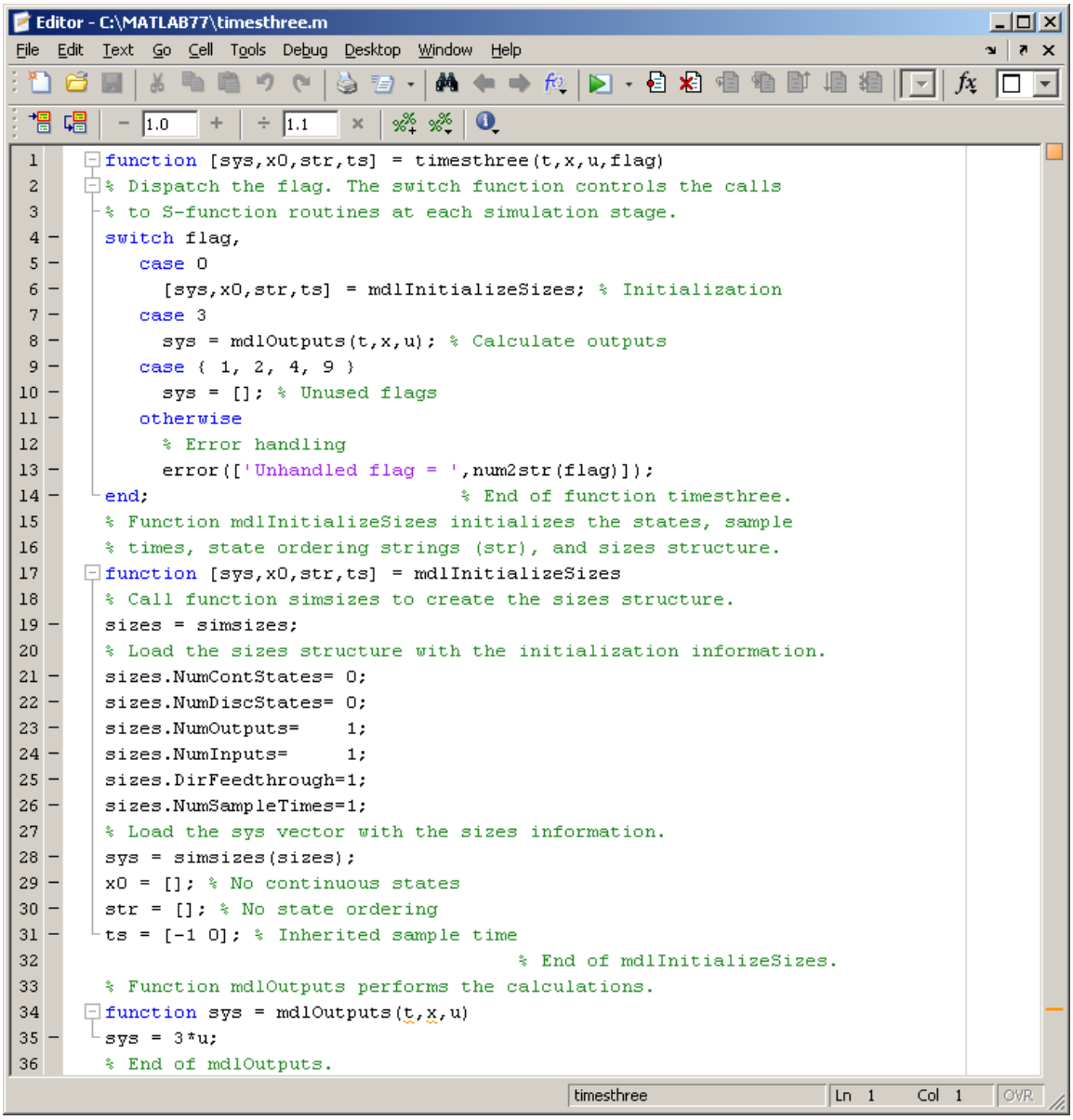
A template implementation of an m-file S-function, *sfuntmpl.m*, is present in the *toolbox>simulink>blocks* location from the top level matlab installation directory or folder. The template consists of a top-level function and a set of skeleton subfunctions, each of which corresponds to a particular value of *flag*. The top-level function invokes the subfunction indicated by *flag*. The subfunctions, which are called S-function callback methods, perform the tasks required of the S-function during simulation. The code for an m-file S-function that follows this standard format can be seen within the Matlab help files at *Product Help> Simulink > Overview of S-Functions > Implementing S-Functions* from the *Help* navigation bar menu. Clicking on the link to *msfuntmpl_basic.m* will display the code.

It is recommended that the structure and naming conventions of the template be followed when creating m-file S-functions.

An example of an m-file S-Function is given below.

Structure of S-Function code m-files

The following is an example that creates a block which takes an input scalar signal and multiplies it by three. The m-file code containing the S-function, shown below, is modeled on the S-function template *sfuntmp1.m* and would have a file name "*timesthree.m*".



Here we deconstruct the S-function file to understand it.

1. The first line specifies the input and output arguments.

```
function [sys,x0,str,ts] = timesthree(t,x,u,flag)
```

2. After the first line, the S-function file is split into the different cases determined by *flag*. As shown in the codes, it only considers case 1 and 3. For cases 1, 2, 4, and 9, it simply sets *sys=[]*. The last two lines of the *timesthree* function are to catch an exceptional case where a bug has occurred during the Simulink run.

3. For case 0 (initialization), the function *mdlInitializeSizes* invokes the *simsizes* command. Without arguments, *simsizes* will create a structure variable which we can then fill with the initial values.

```
sizes = simsizes;
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    1;
sizes.NumInputs=     1;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
```

Using the command *simsizes* again with the structure variable as the argument actually translates the values in the structure *sizes* into a row vector which then gets sent to Simulink via *sys*:

```
sys = simsizes(sizes);
```

It then initializes the states (*x0*), state ordering strings (*str*) and sample times (*ts*).

```
x0 = []; % No continuous states
str = []; % No state ordering
ts = [-1 0]; % Inherited sample time
```

4. For case 3 (output calculations), function *mdlOutputs* performs the calculations:

```
 sys = 3*u;
```

To illustrate this function, consider a time vector

```
>> t = [1 2 3];
```

and a state vector

```
>> x = [1 1];
```

and an input argument

```
>> u = 4;
```

Then the initialization procedure is

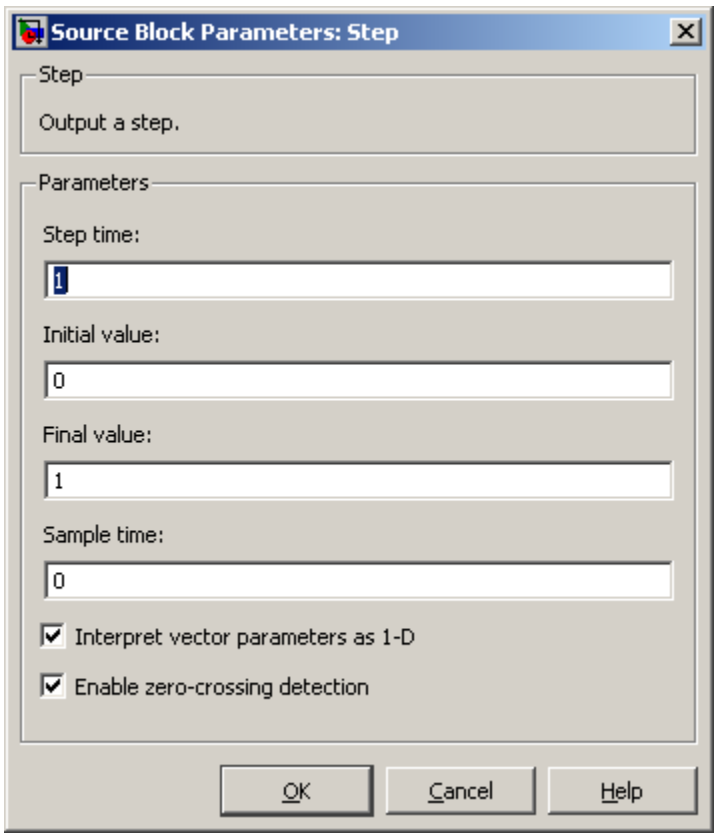
```
>> timesthree(t,x,u,0)

ans =

     0     0     1     1     0     1     1
```

and the calculated output is

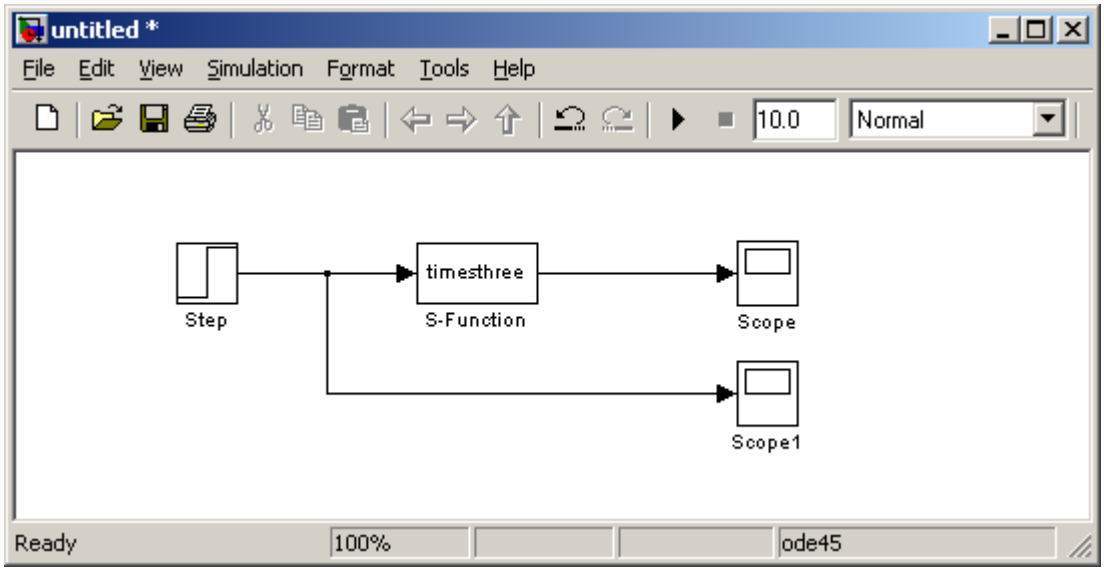
```
>> timesthree(t,x,u,3)

ans =

    12
```

Insertion of an S-function into Simulink

In the Simulink Library browser, go to the [User-Define Functions] subdirectory. Then drag-drop the S-Function block. For this example, connect a *Step* signal to the input of an S-Function block and connect the output of the S-Function block to a *Scope*. Then drag in a second Scope block and connect as a branch from the Step block before input into the S-Function block. Double-click on the Step block to confirm the default parameter values and edit if necessary.



Now double-click on the S-function block to open its dialog box. In that dialog box, change the *Sfunction* name to *timesthree* and leave the parameters empty.



S-Function Test Example

The simulation can now be run and the input and output to the S-function checked from the *Scope1* and *Scope* displays respectively.