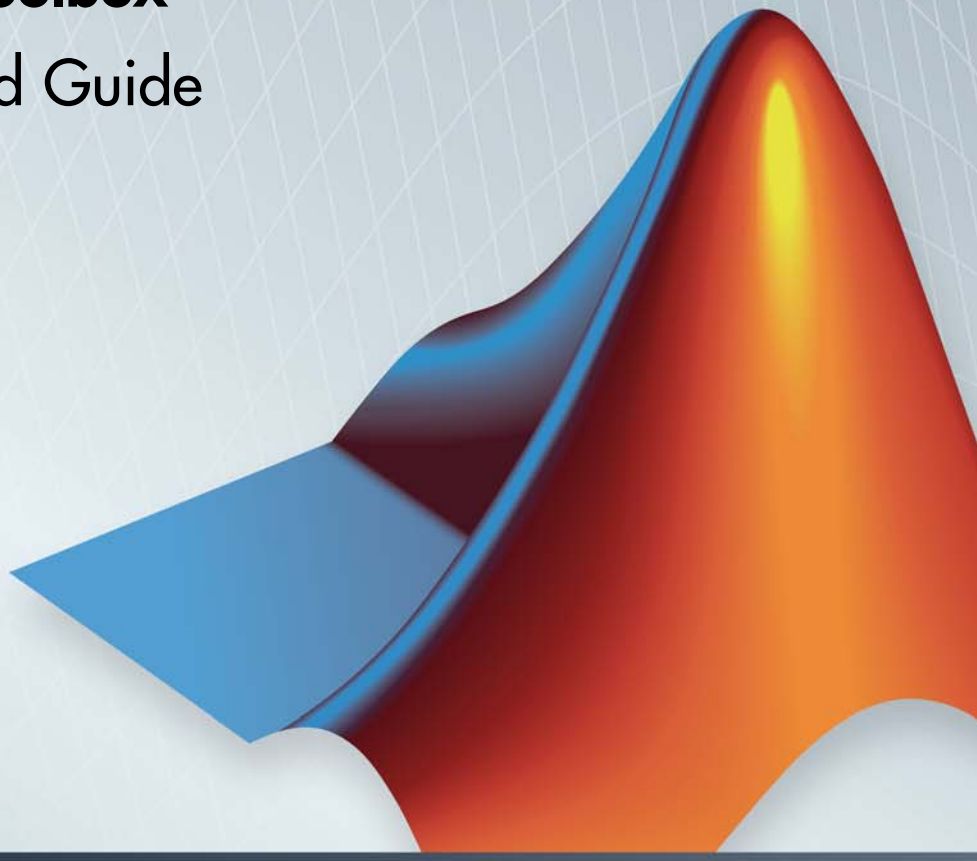
DSP System Toolbox™

Getting Started Guide

R2014**a**



MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

`suggest@mathworks.com`	Product enhancement suggestions
`bugs@mathworks.com`	Bug reports
`doc@mathworks.com`	Documentation error reports
`service@mathworks.com`	Order status, license renewals, passcodes
`info@mathworks.com`	Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

DSP System Toolbox™ Getting Started Guide

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See `www.mathworks.com/trademarks` for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see `www.mathworks.com/patents` for more information.

Revision History

April 2011	First printing	Revised for Version 8.0 (R2011a)
September 2011	Online only	Revised for Version 8.1 (R2011b)
March 2012	Online only	Revised for Version 8.2 (R2012a)
September 2012	Online only	Revised for Version 8.3 (R2012b)
March 2013	Online only	Revised for Version 8.4 (R2013a)
September 2013	Online only	Revised for Version 8.5 (R2013b)
March 2014	Online only	Revised for Version 8.6 (R2014a)

Contents

1 Introduction

DSP System Toolbox Product Description 1-2
Key Features 1-2

Configure Simulink Environment for Signal Processing Models 1-3
Installation 1-3
Required Products 1-3
Related Products 1-4

Configure the Simulink Environment for Signal Processing Models 1-5
Using dspstartup.m 1-5
Settings in dspstartup.m 1-6

2 Design a Filter with fdesign and filterbuilder

Filter Design Process Overview 2-2

Design a Filter Using fdesign 2-4

Design a Filter Using filterbuilder 2-10

3 Design Filters in Simulink

Design and Implement a Filter 3-2

Design a Digital Filter in Simulink 3-2
Add a Digital Filter to Your Model 3-7

Adaptive Filters 3-11
Design an Adaptive Filter in Simulink 3-11
Add an Adaptive Filter to Your Model 3-16
View the Coefficients of Your Adaptive Filter 3-21

4 System Objects

What Is a System Toolbox? 4-2

What Are System Objects? 4-3

When to Use System Objects Instead of MATLAB Functions 4-5
System Objects vs. MATLAB Functions 4-5
Process Audio Data Using Only MATLAB Functions Code 4-5
Process Audio Data Using System Objects 4-6

System Design and Simulation in MATLAB 4-8

System Design and Simulation in Simulink 4-9

System Objects in MATLAB Code Generation 4-10
System Objects in Generated Code 4-10
System Objects in codegen 4-16
System Objects in the MATLAB Function Block 4-16
System Objects in the MATLAB System Block 4-16
System Objects and MATLAB Compiler Software 4-16

System Objects in Simulink 4-17
System Objects in the MATLAB Function Block 4-17
System Objects in the MATLAB System Block 4-17

System Object Methods **4-18**
What Are System Object Methods? **4-18**
The Step Method **4-18**
Common Methods **4-19**

System Design in MATLAB Using System Objects **4-21**
Create Components for Your System **4-21**
Configure Components for Your System **4-22**
Assemble Components to Create Your System **4-23**
Run Your System **4-25**
Reconfigure Your System During Runtime **4-25**

System Design in Simulink Using System Objects **4-28**
Define New Kinds of System Objects for Use in Simulink . . **4-28**
Test New System Objects in MATLAB **4-34**
Add System Objects to Your Simulink Model **4-35**

1

Introduction

- "DSP System Toolbox Product Description" on page 1-2
- "Configure Simulink Environment for Signal Processing Models" on page 1-3
- "Configure the Simulink Environment for Signal Processing Models" on page 1-5

DSP System Toolbox Product Description

Design and simulate signal processing systems

DSP System Toolbox™ provides algorithms for designing and simulating signal processing systems. These capabilities are provided as MATLAB® functions, MATLAB System objects, and Simulink® blocks. The system toolbox includes design methods for specialized FIR and IIR filters, FFTs, multirate processing, and DSP techniques for processing streaming data and creating real-time prototypes. You can design adaptive and multirate filters, implement filters using computationally efficient architectures, and simulate floating-point digital filters. Tools for signal I/O from files and devices, signal generation, spectral analysis, and interactive visualization enable you to analyze system behavior and performance. For rapid prototyping and embedded system design, the system toolbox supports fixed-point arithmetic and C or HDL code generation.

Key Features

- Algorithms available as MATLAB System objects and Simulink blocks
- Simulation of streaming, frame-based, and multirate systems
- Signal generators and I/O support for multimedia files and devices, including ASIO™ drivers and multichannel audio
- Design methods for specialized filters, including parametric equalizers and adaptive, multirate, octave, and acoustic weighting filters
- Filter realization architectures, including second-order sections and lattice wave digital filters
- Signal measurements for peak-to-peak, peak-to-RMS, state-level estimation, and bilevel waveform metrics
- FFT, spectral estimation, windowing, signal statistics, and linear algebra
- Algorithm support for floating-point, integer, and fixed-point data types
- Support for fixed-point modeling and C and HDL code generation

Configure Simulink Environment for Signal Processing Models

In this section...
"Installation" on page 1-3
"Required Products" on page 1-3
"Related Products" on page 1-4

Installation

Before you begin working, you need to install the product on your computer.

Installing the DSP System Toolbox Software

The DSP System Toolbox software follows the same installation procedure as the MATLAB toolboxes.

Installing Online Documentation

Installing the documentation is part of the installation process:

- Installation from a DVD — Start the MathWorks® installer. When prompted, select the **Product** check boxes for the products you want to install. The documentation is installed along with the products.
- Installation from a Web download — If you update the DSP System Toolbox software using a Web download and you want to view the documentation with the MATLAB Help browser, you must install the documentation on your hard drive.

  Download the files from the Web. Then, start the installer, and select the **Product** check boxes for the products you want to install. The documentation is installed along with the products.

Required Products

The DSP System Toolbox product is part of a family of MathWorks products. You need to install several products to use the toolbox. For more

information about the required products, see the MathWorks Web site, at `http://www.mathworks.com/products/dsp-system/requirements.html`.

Related Products

MathWorks provides several products that are relevant to the kinds of tasks you can perform with DSP System Toolbox software.

For more information about any of these products, see either

- The online documentation for that product if it is installed on your system
- The MathWorks Web site, at `http://www.mathworks.com/products/dsp-system/related.html`.

Configure the Simulink Environment for Signal Processing Models

In this section...

"Using dspstartup.m" on page 1-5

"Settings in dspstartup.m" on page 1-6

Using dspstartup.m

The DSP System Toolbox product provides a file, `dspstartup.m`, that lets you automatically configure the Simulink environment for signal processing simulation. We recommend these configuration parameters for models that contain DSP System Toolbox blocks. Because these blocks calculate values directly rather than solving differential equations, you must configure the Simulink solver to behave like a scheduler. The solver, while in scheduler mode, uses a block sample time to determine when the code behind each block executes. For example, if the sample time of a Sine Wave block is `0.05`, the solver executes the code behind this block and every other block with this sample time once every 0.05 seconds.

Note When working with models that contain DSP System Toolbox blocks, use source blocks that allow you to specify a sample time. When your source block does not have a **Sample time** parameter, you must add a Zero-Order Hold block in your model and use it to specify the sample time. For more information, see "Continuous-Time Source Blocks". The exception to this rule is the Constant block, which can have a constant sample time. When it does, Simulink executes this block and records the constant value once, which allows for faster simulations and more compact generated code.

To use the `dspstartup` file to configure Simulink for signal processing simulations, you can

- Type `dspstartup` at the MATLAB command line. All new models have settings customized for signal processing applications. Existing models are not affected.

- Place a call to `dspstartup` within the `startup.m` file. This is an efficient way to use `dspstartup` if you want these settings to be in effect every time you start Simulink. For more information about performing automated tasks at startup, see the documentation for the `startup` command in the MATLAB Function Reference.

The `dspstartup` file executes the following commands:

```
set_param(0, ...
          'SingleTaskRateTransMsg','error', ...
          'multiTaskRateTransMsg', 'error', ...
          'Solver',                'fixedstepdiscrete', ...
          'SolverMode',            'SingleTasking', ...
          'StartTime',             '0.0', ...
          'StopTime',              'inf', ...
          'FixedStep',             'auto', ...
          'SaveTime',              'off', ...
          'SaveOutput',            'off', ...
          'AlgebraicLoopMsg',      'error', ...
          'SignalLogging',         'off');
```

You can edit the `dspstartup` file to change any of these settings or to add your own custom settings. For complete information about these settings, see "Model Parameters" in the Simulink documentation.

Settings in dspstartup.m

A number of the settings in the `dspstartup` file are chosen to improve the performance of the simulation:

- `'Solver'` is set to `'fixedstepdiscrete'`.

  This selects the fixed-step solver option instead of the Simulink default variable-step solver. This mode enables code generation from the model using the Simulink Coder™ product.

- `'Stop time'` is set to `'Inf'`.

  The simulation runs until you manually stop it by selecting **Stop** from the **Simulation** menu.

- `'SaveTime'` is set to `'off'`.

Simulink does not save the `tout` time-step vector to the workspace. The time-step record is not usually needed for analyzing discrete-time simulations, and disabling it saves a considerable amount of memory, especially when the simulation runs for an extended time.

- `'SaveOutput'` is set to `'off'`.

  Simulink Outport blocks in the top level of a model do not generate an output (`yout`) in the workspace.

2

Design a Filter with fdesign and filterbuilder

- "Filter Design Process Overview" on page 2-2
- "Design a Filter Using fdesign" on page 2-4
- "Design a Filter Using filterbuilder" on page 2-10

Filter Design Process Overview

Note You must have the Signal Processing Toolbox™ installed to use `fdesign` and `filterbuilder`. Advanced capabilities are available if your installation additionally includes the DSP System Toolbox license. You can verify the presence of both toolboxes by typing `ver` at the command prompt.

Filter design through user-defined specifications is the core of the `fdesign` approach. This specification-centric approach places less emphasis on the choice of specific filter algorithms, and more emphasis on performance during the design a good working filter. For example, you can take a given set of design parameters for the filter, such as a stopband frequency, a passband frequency, and a stopband attenuation, and— using these parameters— design a specification object for the filter. You can then implement the filter using this specification object. Using this approach, it is also possible to compare different algorithms as applied to a set of specifications.

There are two distinct objects involved in filter design:

- Specification Object — Captures the required design parameters of a filter
- Implementation Object — Describes the designed filter; includes the array of coefficients and the filter structure

The distinction between these two objects is at the core of the filter design methodology. The basic attributes of each of these objects are outlined in the following table.

Specification Object	Implementation Object
High-level specification	Filter coefficients
Algorithmic properties	Filter structure

You can run the code in the following examples from the Help browser (select the code, right-click the selection, and choose **Evaluate Selection** from the context menu), or you can enter the code on the MATLAB command line. Before you begin this example, start MATLAB and verify that you have installed the Signal Processing Toolbox software. If you wish to access the

full functionality of `fdesign` and `filterbuilder`, you should additionally obtain the DSP System Toolbox software. You can verify the presence of these products by typing `ver` at the command prompt.

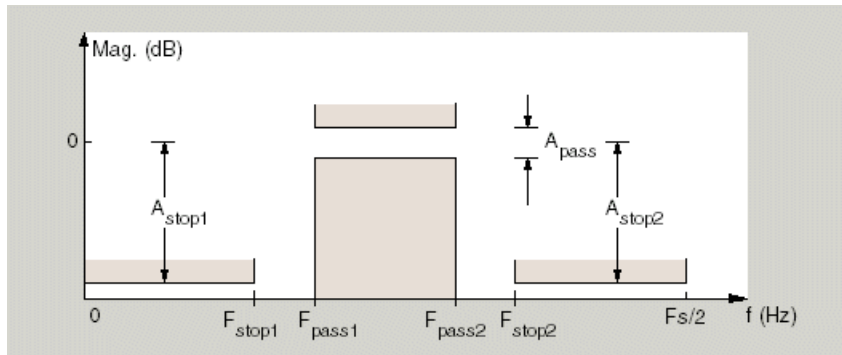
Design a Filter Using fdesign

Use the following two steps to design a simple filter.

1. Create a filter specification object.
2. Design your filter.

Design a Filter in Two Steps

Assume that you want to design a bandpass filter. Typically a bandpass filter is defined as shown in the following figure.



In this example, a sampling frequency of `Fs` = 48 kHz is used. This bandpass filter has the following specifications, specified here using MATLAB code:

```
A_stop1 = 60;  % Attenuation in the first stopband = 60 dB
F_stop1 = 8400;  % Edge of the stopband = 8400 Hz
F_pass1 = 10800; % Edge of the passband = 10800 Hz
F_pass2 = 15600; % Closing edge of the passband = 15600 Hz
F_stop2 = 18000; % Edge of the second stopband = 18000 Hz
A_stop2 = 60;  % Attenuation in the second stopband = 60 dB
A_pass = 1;  % Amount of ripple allowed in the passband = 1 dB
```

In the following two steps, these specifications are passed to the `fdesign.bandpass` method as parameters.

Step 1

To create a filter specification object, evaluate the following code at the MATLAB prompt:

```
d = fdesign.bandpass
```

Now, pass the filter specifications that correspond to the default `Specification` — `fst1`,`fp1`,`fp2`,`fst2`,`ast1`,`ap`,`ast2`. This example adds `fs` as the final input argument to specify the sampling frequency of 48 kHz.

```
>> BandPassSpecObj = ...
   fdesign.bandpass('Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2', ...
  F_stop1, F_pass1, F_pass2, F_stop2, A_stop1, A_pass, ...
  A_stop2, 48000)
```

Note The order of the filter is not specified, allowing a degree of freedom for the algorithm design in order to achieve the specification. The design will be a minimum order design.

The specification parameters, such as `Fstop1`, are all given default values when none are provided. You can change the values of the specification parameters after the filter specification object has been created. For example, if there are two values that need to be changed, `Fpass2` and `Fstop2`, use the `set` command, which takes the object first, and then the parameter value pairs. Evaluate the following code at the MATLAB prompt:

```
>> set(BandPassSpecObj, 'Fpass2', 15800, 'Fstop2', 18400)
```

`BandPassSpecObj` is the new filter specification object which contains all the required design parameters, including the filter type.

You may also change parameter values in filter specification objects by accessing them as if they were elements in a `struct` array.

```
>> BandPassSpecObj.Fpass2=15800;
```

Step 2

Design the filter by using the `design` command. You can access the design methods available for you specification object by calling the `designmethods` function. For example, in this case, you can execute the command

```
>> designmethods(BandPassSpecObj)


Design Methods for class
fdesign.bandpass (Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2):


butter
cheby1
cheby2
ellip
equiripple
kaiserwin
```

After choosing a design method use, you can evaluate the following at the MATLAB prompt (this example assumes you've chosen '`equiripple`'):

```
>> BandPassFilt = design(BandPassSpecObj, 'equiripple')

BandPassFilt =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x44 double]
    PersistentMemory: false
```

If you have the DSP System Toolbox installed, you can also design your filter with a filter System object™. To create a filter System object with the same specification object `BandPassSpecObj`, you can execute the commands

```
>> designmethods(BandPassSpecObj,...
'SystemObject',true)
```

```
Design Methods that support System objects for class
fdesign.bandpass (Fst1,Fp1,Fp2,Fst2,Ast1,Ap,Ast2):


butter
cheby1
cheby2
ellip
equiripple
kaiserwin


>> BandPassFiltSysObj = design(BandPassSpecObj,...
'equiripple','SystemObject',true)


  System: dsp.FIRFilter

  Properties:
              Structure: 'Direct form'
        NumeratorSource: 'Property'
              Numerator: [1x44 double]
      InitialConditions: 0
    FrameBasedProcessing: true

  Show fixed-point properties
```

Available design methods and design options for filter System objects are not necessarily the same as those for filter objects.

Note If you do not specify a design method, a default method will be used. For example, you can execute the command

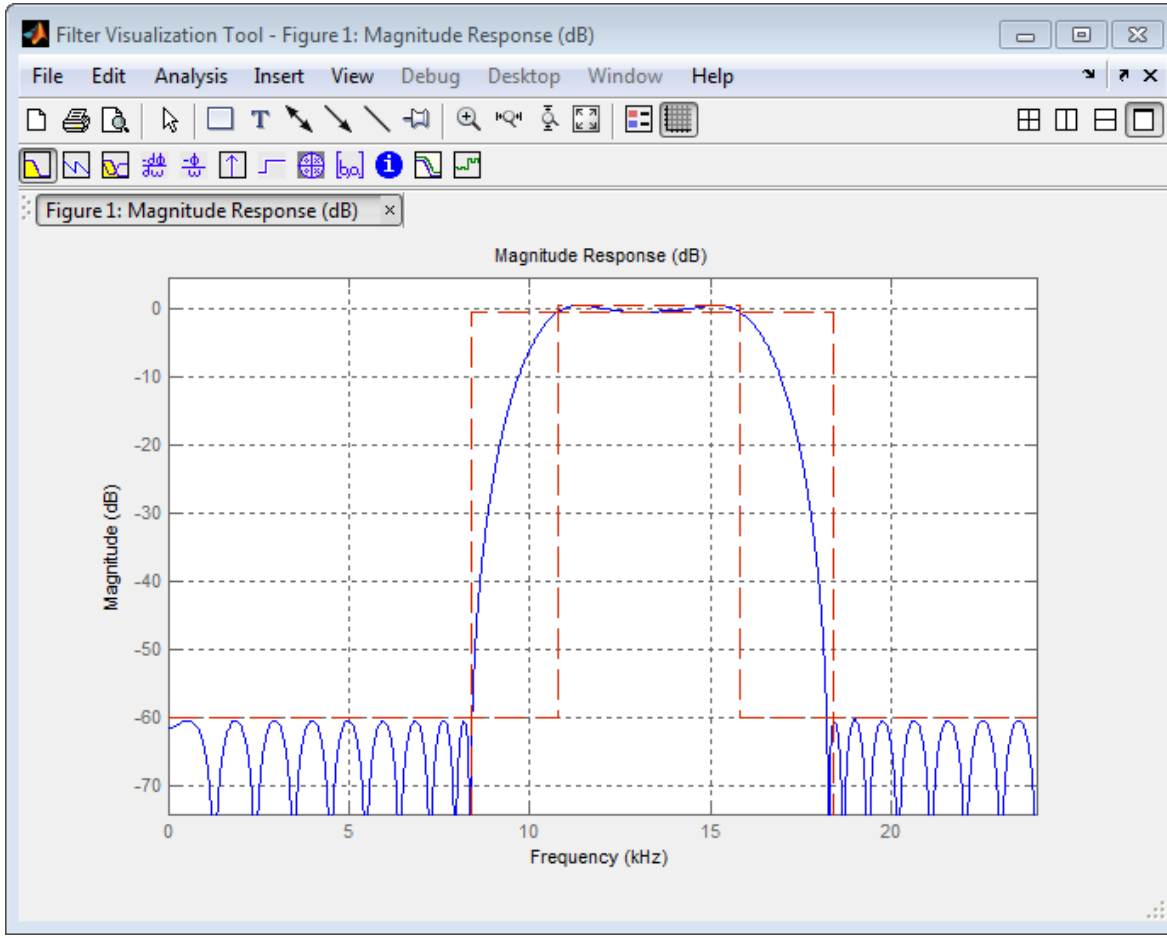
```
>> BandPassFilt = design(BandPassSpecObj)

BandPassFilt =

     FilterStructure: 'Direct-Form FIR'
          Arithmetic: 'double'
           Numerator: [1x44 double]
    PersistentMemory: false
```

and a design method will be selected automatically.

To check your work, you can plot the filter magnitude response using the Filter Visualization tool. Verify that all the design parameters are met:

```
>> fvtool(BandPassFilt) %plot the filter magnitude response
```

If you have the DSP System Toolbox installed, the Filter Visualization tool produces the following figure with the dashed red lines indicating the transition bands and unity gain (0 in dB) over the passband.


Filter Visualization Tool - Figure 1: Magnitude Response (dB)
File Edit Analysis Insert View Debug Desktop Window Help
Figure 1: Magnitude Response (dB)
Magnitude Response (dB)
Magnitude (dB)
0
-10
-20
-30
-40
-50
-60
-70
0
5
10
15
20
Frequency (kHz)

Design a Filter Using filterbuilder

Filterbuilder presents the option of designing a filter using a GUI dialog box as opposed to the command line instructions. You can use Filterbuilder to design the same bandpass filter designed in the previous section, "Design a Filter Using fdesign" on page 2-4

Design a Simple Filter in Filterbuilder

To design the filter using the Filterbuilder GUI:

1. Type the following at the MATLAB prompt:

   ```
   filterbuilder
   ```

2. Select `Bandpass` filter response from the list in the dialog box, and hit the **OK** button.

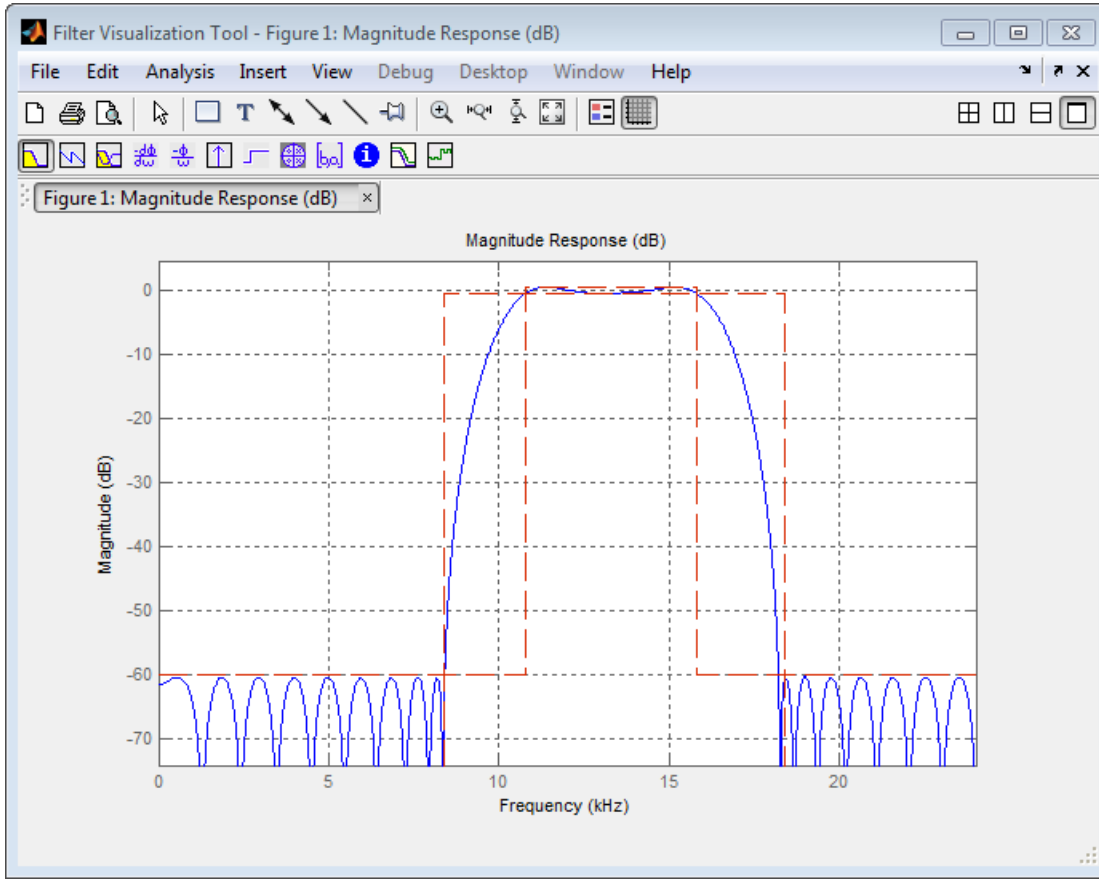
3. Enter the correct frequencies for **Fpass2** and **Fstop2**, then click **OK**. Here the specification uses normalized frequency, so that the passband and stopband edges are expressed as a fraction of the Nyquist frequency (in this case, 48/2 kHz). The following message appears at the MATLAB prompt:

   ```
   The variable 'Hbp' has been exported to the command window.
   ```

   If you display the Workspace tab, you see the object `Hbp` has been placed on your workspace.

4. To check your work, plot the filter magnitude response using the Filter Visualization tool. Verify that all the design parameters are met:

   ```
   fvtool(Hbp) %plot the filter magnitude response
   ```



Note that the dashed red lines on the preceding figure will only appear if you are using the DSP System Toolbox software.

3

Design Filters in Simulink

- "Design and Implement a Filter" on page 3-2
- "Adaptive Filters" on page 3-11

Design and Implement a Filter

In this section...

"Design a Digital Filter in Simulink" on page 3-2

"Add a Digital Filter to Your Model" on page 3-7

Design a Digital Filter in Simulink

You can design lowpass, highpass, bandpass, and bandstop filters using either the Digital Filter Design block or the Filter Realization Wizard. These blocks are capable of calculating filter coefficients for various filter structures. In this section, you use the Digital Filter Design block to convert white noise to low frequency noise so you can simulate its effect on your system.

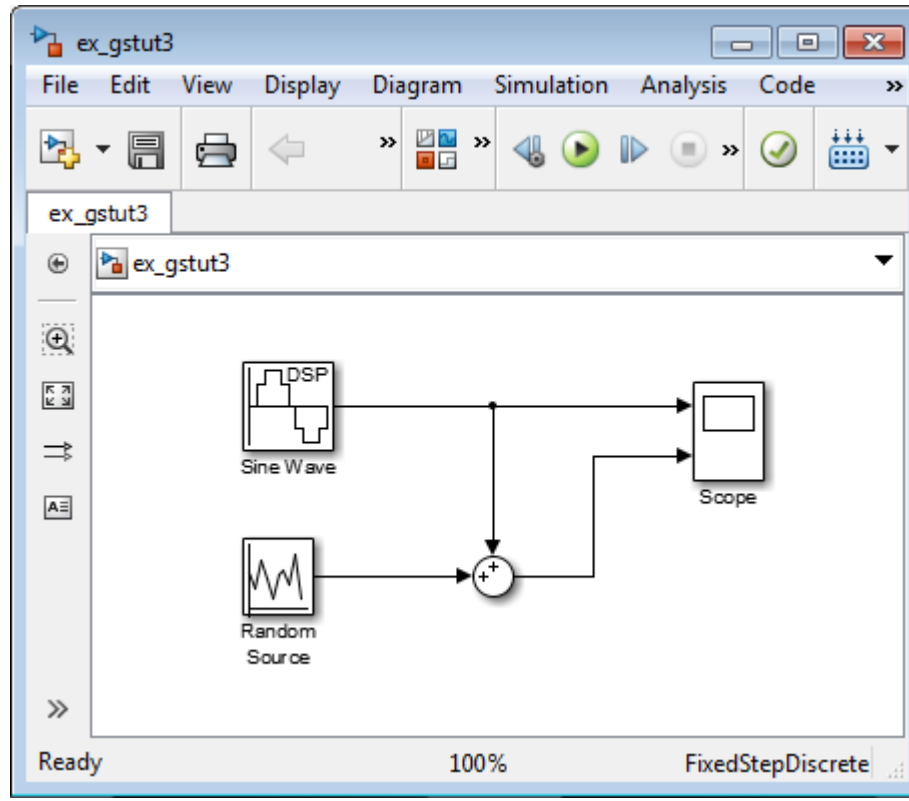
As a practical application, suppose a pilot is speaking into a microphone within the cockpit of an airplane. The noise of the wind passing over the fuselage is also reaching the microphone. A sensor is measuring the noise of the wind on the outside of the plane. You want to estimate the wind noise inside the cockpit and subtract it from the input to the microphone so that only the pilot's voice is transmitted. In this chapter, you first learn how to model the low frequency noise that is reaching the microphone. Later, you learn how to remove this noise so that only the pilot's voice is heard.

In this topic, you use a Digital Filter Design block to create low frequency noise, which models the wind noise inside the cockpit:

1. Open the model by typing

   ```
   ex_gstut3
   ```

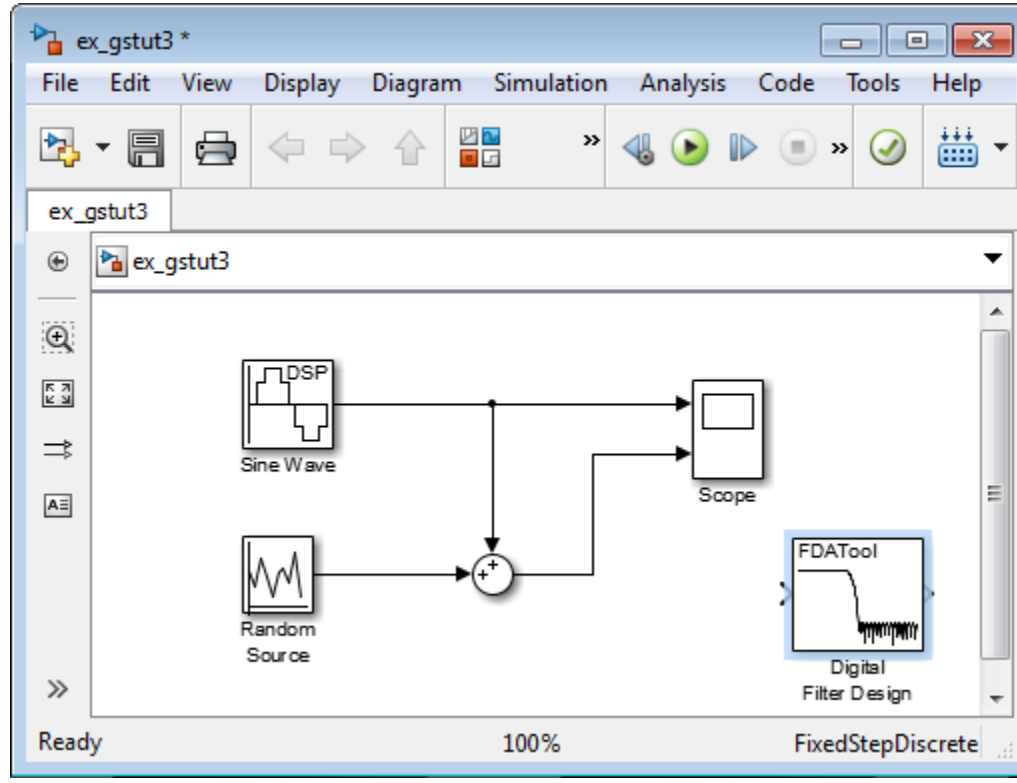
   at the MATLAB command prompt. This model contains a Scope block that displays the original sine wave and the sine wave with white noise added.



2 Open the DSP System Toolbox library by typing `dsplib` at the MATLAB command prompt.

3 Convert white noise to low frequency noise by introducing a Digital Filter Design block into your model. In the airplane scenario, the air passing over the fuselage creates white noise that is measured by a sensor. The Random Source block models this noise. The fuselage of the airplane converts this white noise to low frequency noise, a type of colored noise, which is heard inside the cockpit. This noise contains only certain frequencies and is more difficult to eliminate. In this example, you model the low frequency noise using a Digital Filter Design block. This block uses the functionality of the Filter Design and Analysis Tool (FDATool) to design a filter.

Double-click the Filtering library, and then double-click the Filter Implementations sublibrary. Click-and-drag the Digital Filter Design block into your model.



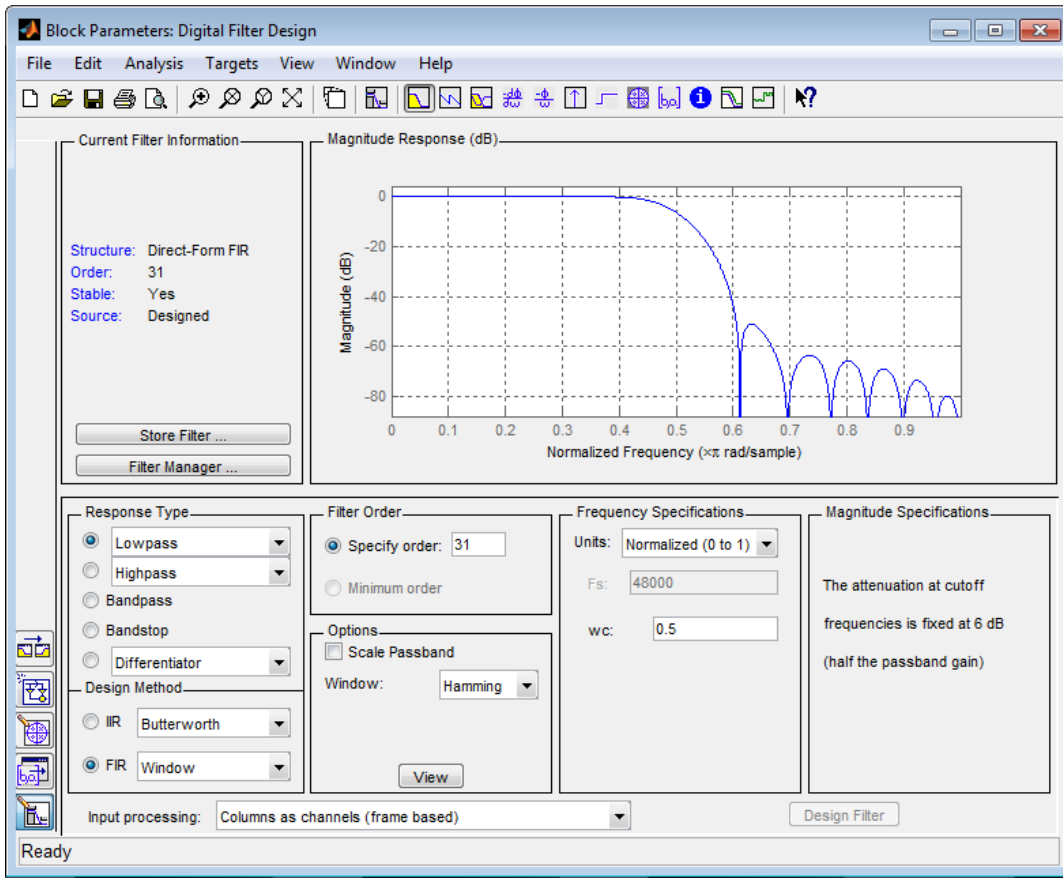
4 Set the Digital Filter Design block parameters to design a lowpass filter and create low frequency noise. Open the block parameters dialog box by double-clicking the block. Set the parameters as follows:

- **Response Type** = `Lowpass`
- **Design Method** = **FIR** and, from the list, choose `Window`
- **Filter Order** = **Specify order** and enter `31`
- **Scale Passband** — Cleared
- **Window** = `Hamming`

- **Units** = `Normalized (0 to 1)`
- **wc** = `0.5`

Based on these parameters, the Digital Filter Design block designs a lowpass FIR filter with 32 coefficients and a cutoff frequency of 0.5. The block multiplies the time-domain response of your filter by a 32 sample Hamming window.

5 Click **Design Filter** at the bottom center of the dialog box to view the magnitude response of your filter in the **Magnitude Response** pane. The Digital Filter Design dialog box should now look similar to the following figure.



You have now designed a digital lowpass filter using the Digital Filter Design block.

You can experiment with the Digital Filter Design block in order to design a filter of your own. For more information on the block functionality, see the Digital Filter Design block reference page. For more information on the Filter Design and Analysis Tool, see “FDATool” in the Signal Processing Toolbox documentation.

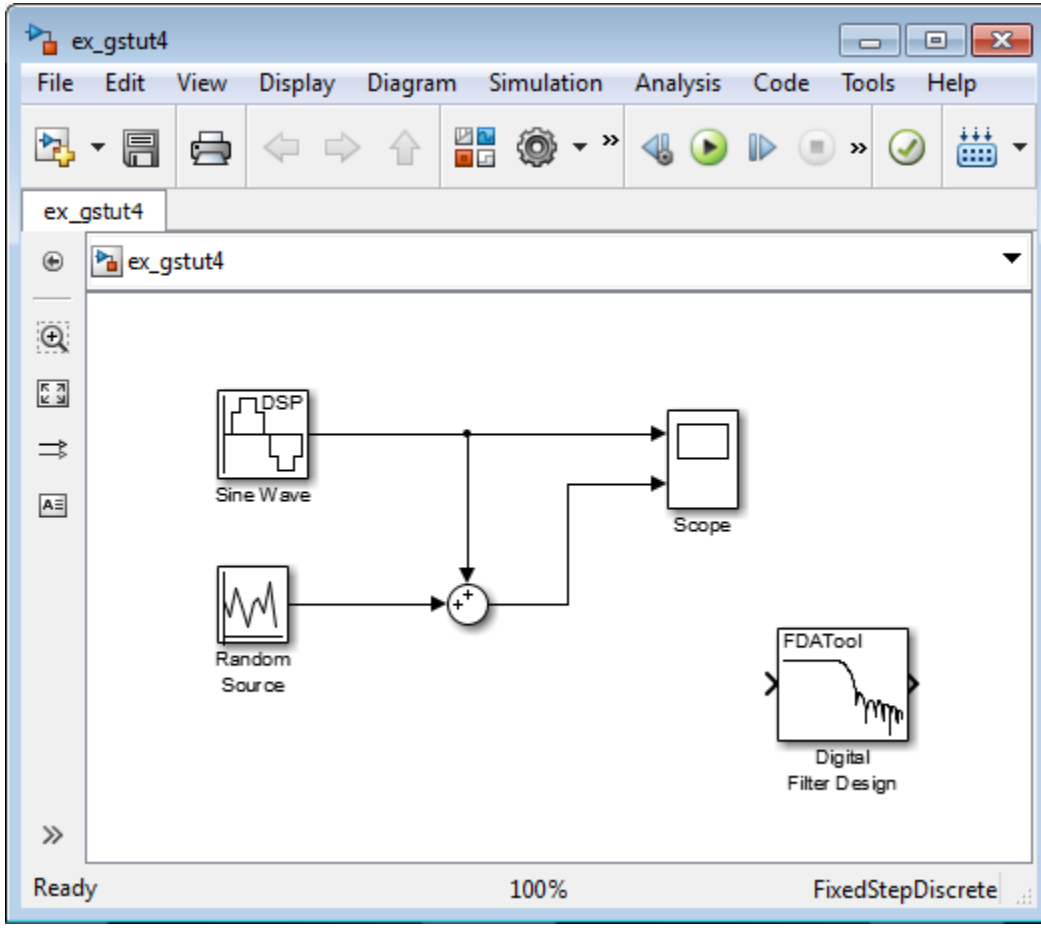
Add a Digital Filter to Your Model

In this topic, you add the lowpass filter you designed in "Design a Digital Filter in Simulink" on page 3-2 to your block diagram. Use this filter, which converts white noise to colored noise, to simulate the low frequency wind noise inside the cockpit:

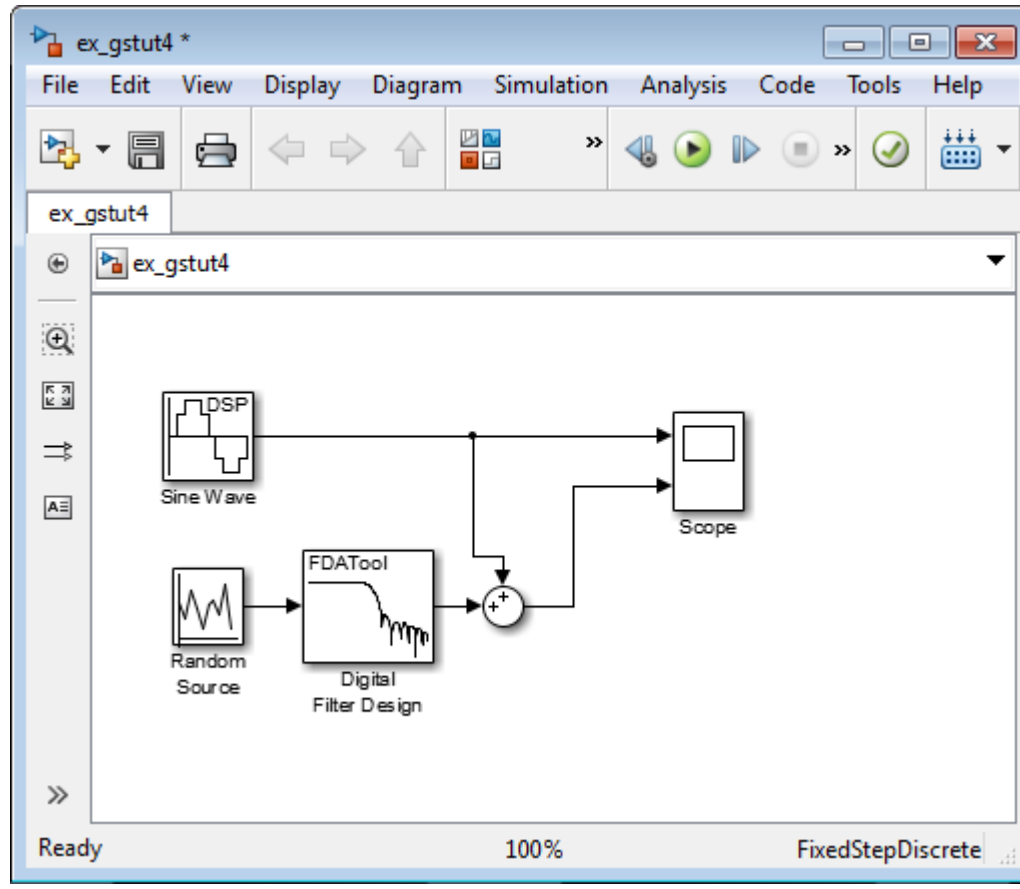
1. If the model you created in "Design a Digital Filter in Simulink" on page 3-2 is not open on your desktop, you can open an equivalent model by typing

   ```
   ex_gstut4
   ```

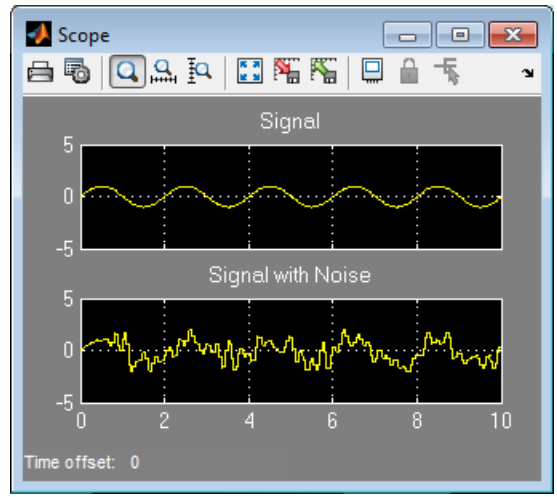
   at the MATLAB command prompt.



2 Incorporate the Digital Filter Design block into your block diagram by placing it between the Random Source block and the Sum block.



3 Run your model and view the results in the Scope window. This window shows the original input signal and the signal with low frequency noise added to it.



You have now built a digital filter and used it to model the presence of colored noise in your signal. This is analogous to modeling the low frequency noise reaching the microphone in the cockpit of the aircraft. Now that you have added noise to your system, you can experiment with methods to eliminate it.

Adaptive Filters

In this section...

"Design an Adaptive Filter in Simulink" on page 3-11

"Add an Adaptive Filter to Your Model" on page 3-16

"View the Coefficients of Your Adaptive Filter" on page 3-21

Design an Adaptive Filter in Simulink

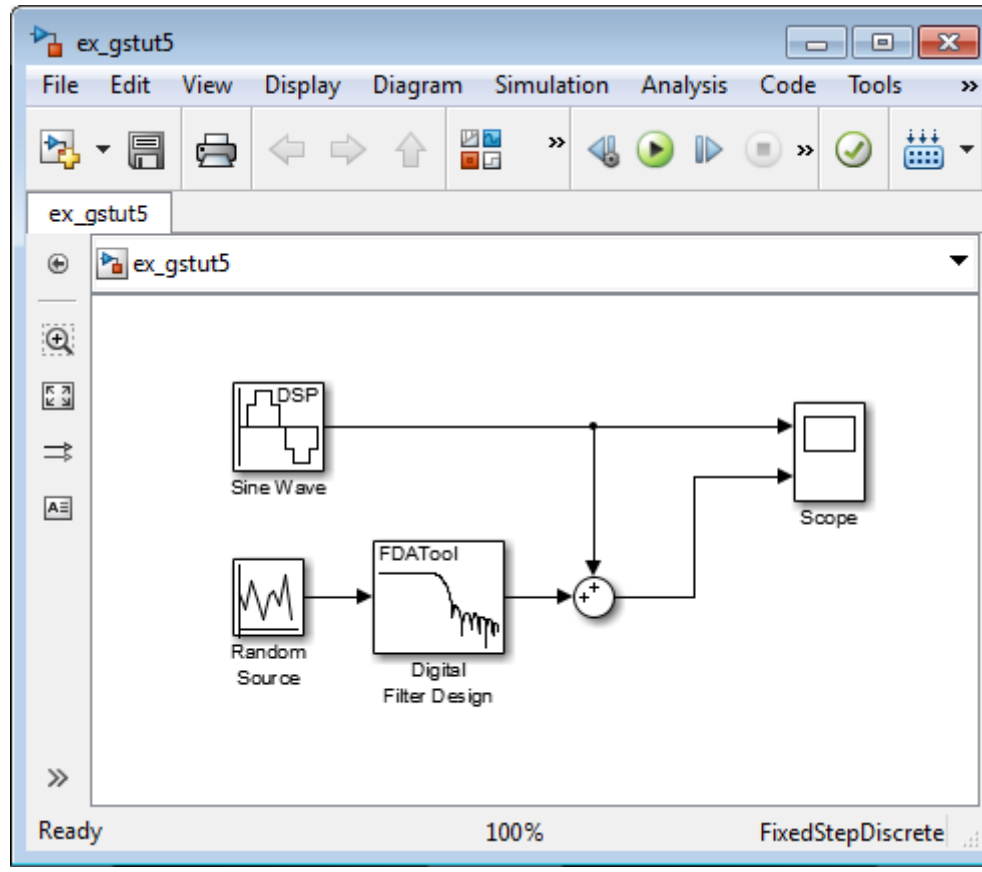
Adaptive filters track the dynamic nature of a system and allow you to eliminate time-varying signals. The DSP System Toolbox libraries contain blocks that implement least-mean-square (LMS), Block LMS, Fast Block LMS, and recursive least squares (RLS) adaptive filter algorithms. These filters minimize the difference between the output signal and the desired signal by altering their filter coefficients. Over time, the adaptive filter's output signal more closely approximates the signal you want to reproduce.

In this topic, you design an LMS adaptive filter to remove the low frequency noise in your signal:

1. If the model you created in "Add a Digital Filter to Your Model" on page 3-7 is not open on your desktop, you can open an equivalent model by typing

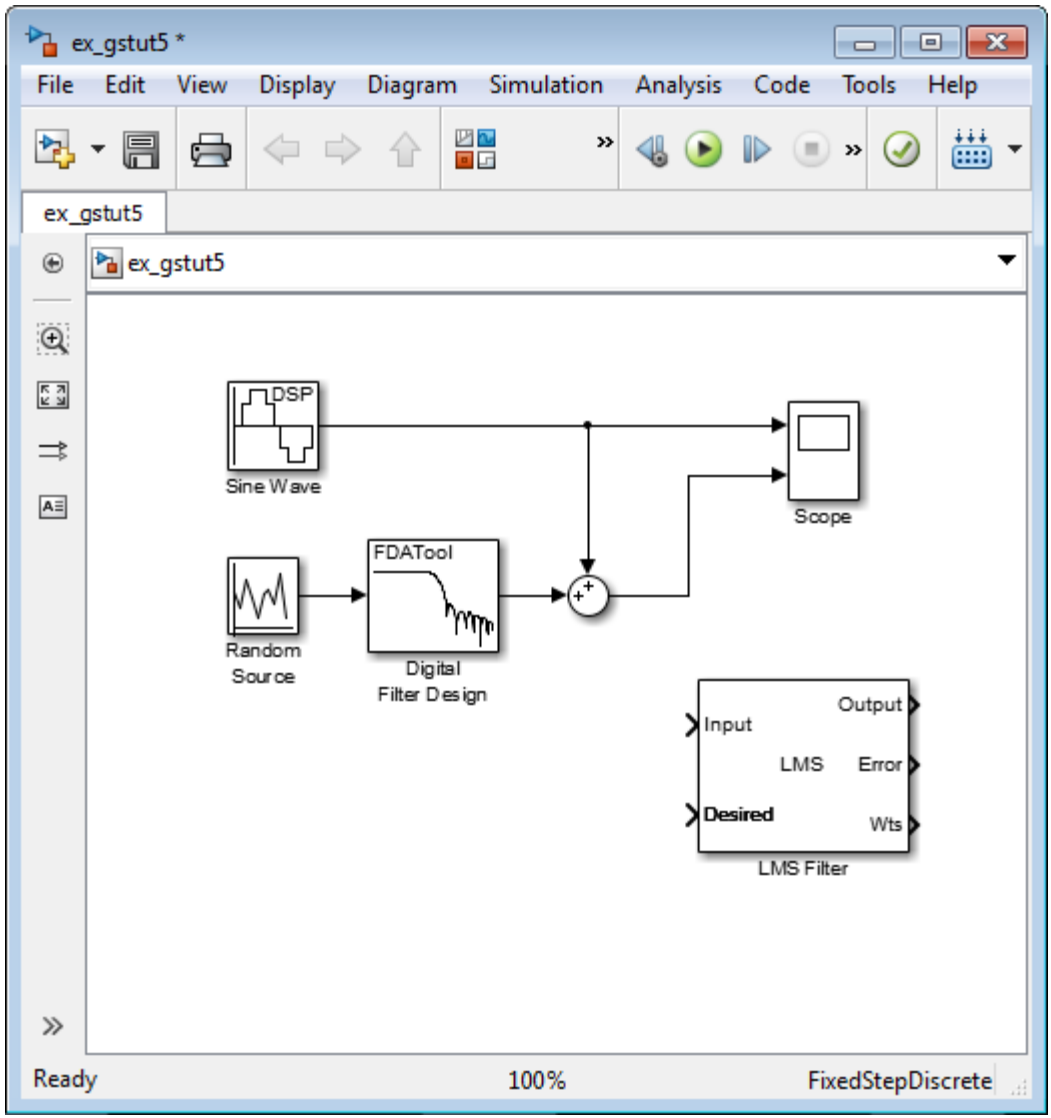
   ```
   ex_gstut5
   ```

   at the MATLAB command prompt.



2 Open the DSP System Toolbox library by typing `dsplib` at the MATLAB command prompt.

3 Remove the low frequency noise from your signal by adding an LMS Filter block to your system. In the airplane scenario, this is equivalent to subtracting the wind noise inside the cockpit from the input to the microphone. Double-click the Filtering sublibrary, and then double-click the Adaptive Filters library. Add the LMS Filter block into your model.

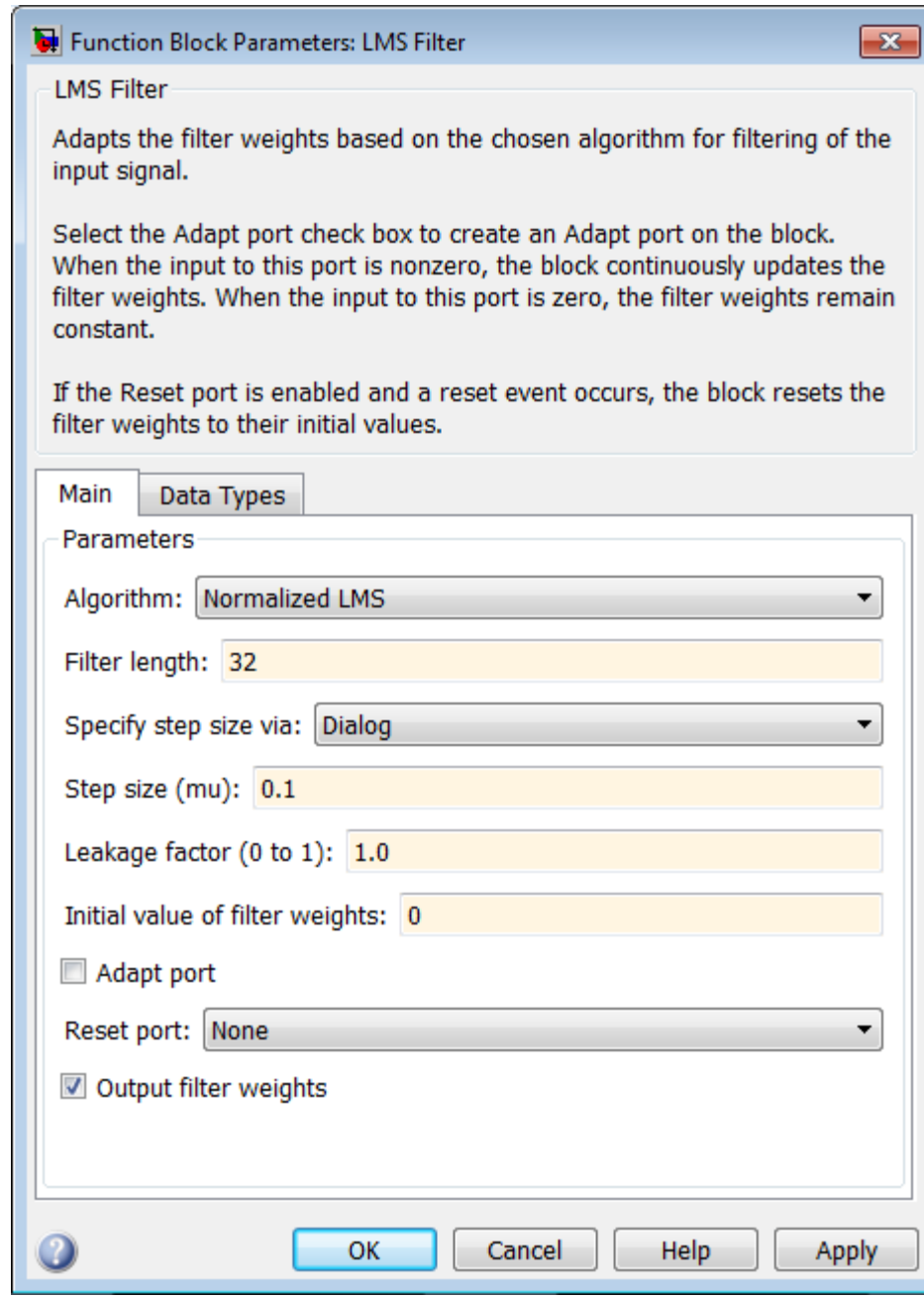


4 Set the LMS Filter block parameters to model the output of the Digital Filter Design block. Open its dialog box by double-clicking the block. Set the block parameters as follows:

- **Algorithm** = `Normalized LMS`

- **Filter length** = `32`
- **Specify step size via** = `Dialog`
- **Step size (mu)** = `0.1`
- **Leakage factor (0 to 1)** = `1.0`
- **Initial value of filter weights** = `0`
- Clear the **Adapt port** check box.
- **Reset port** = `None`
- Select the **Output filter weights** check box.

  The LMS Filter dialog box should now look like the following figure:


Function Block Parameters: LMS Filter
LMS Filter
Adapts the filter weights based on the chosen algorithm for filtering of the input signal.
Select the Adapt port check box to create an Adapt port on the block. When the input to this port is nonzero, the block continuously updates the filter weights. When the input to this port is zero, the filter weights remain constant.
If the Reset port is enabled and a reset event occurs, the block resets the filter weights to their initial values.
Main
Data Types
Parameters
Algorithm: Normalized LMS
Filter length: 32
Specify step size via: Dialog
Step size (mu): 0.1
Leakage factor (0 to 1): 1.0
Initial value of filter weights: 0
Adapt port
Reset port: None
Output filter weights
OK
Cancel
Help
Apply

5 Click **Apply**.

Based on these parameters, the LMS Filter block computes the filter weights using the normalized LMS equations. The filter order you specified is the same as the filter order of the Digital Filter Design block. The **Step size (mu)** parameter defines the granularity of the filter update steps. Because you set the **Leakage factor (0 to 1)** parameter to `1.0`, the current filter coefficient values depend on the filter's initial conditions and all of the previous input values. The initial value of the filter weights (coefficients) is zero. Since you selected the **Output filter weights** check box, the Wts port appears on the block. The block outputs the filter weights from this port.

Now that you have set the block parameters of the LMS Filter block, you can incorporate this block into your block diagram.

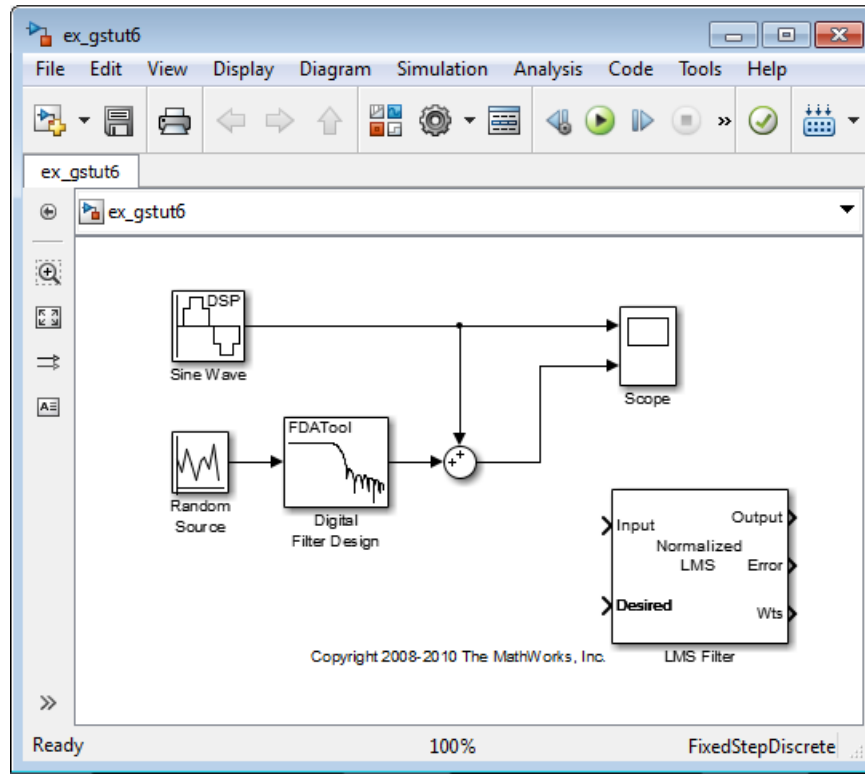
Add an Adaptive Filter to Your Model

In this topic, you recover your original sinusoidal signal by incorporating the adaptive filter you designed in "Design an Adaptive Filter in Simulink" on page 3-11 into your system. In the aircraft scenario, the adaptive filter models the low frequency noise heard inside the cockpit. As a result, you can remove the noise so that the pilot's voice is the only input to the microphone:

1 If the model you created in "Design an Adaptive Filter in Simulink" on page 3-11 is not open on your desktop, you can open an equivalent model by typing

```
ex_gstut6
```

at the MATLAB command prompt.



2 Add a Sum block to your model to subtract the output of the adaptive filter from the sinusoidal signal with low frequency noise. From the Simulink Math Operations library, drag a Sum block into your model. Open the Sum dialog box by double-clicking this block. Change the **List of signs** parameter to `|+-` and then click **OK**.

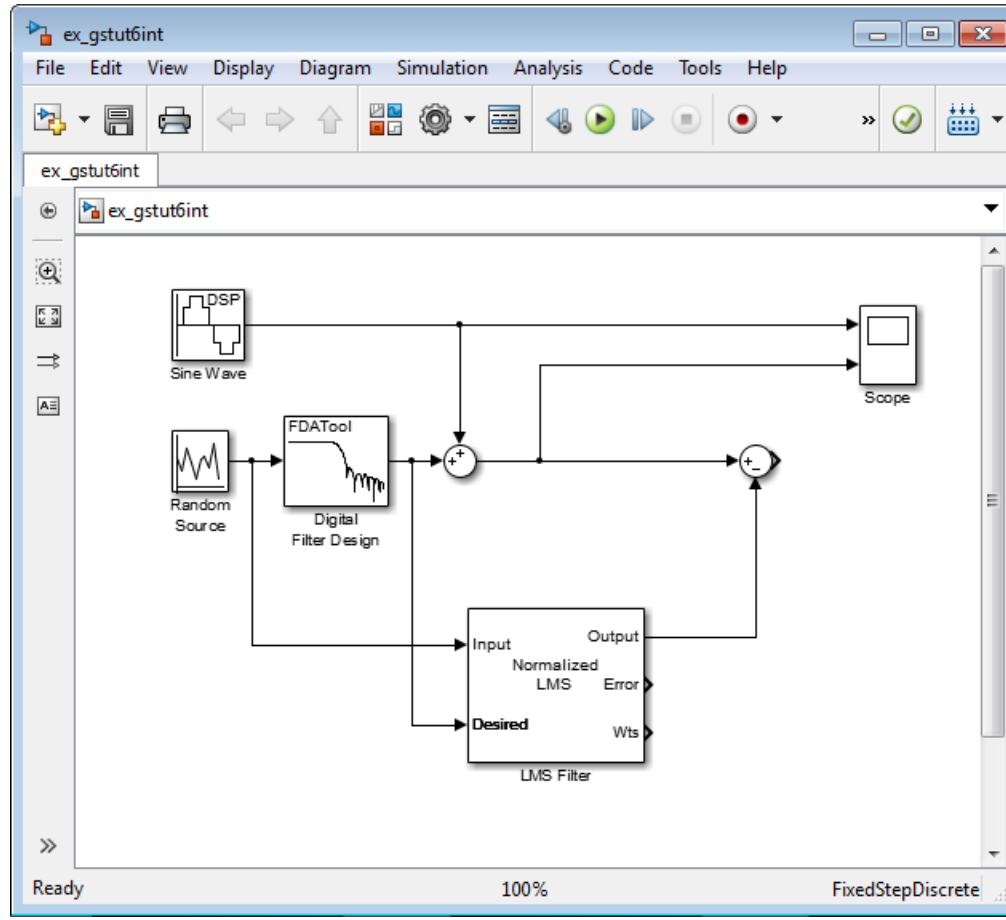
3 Incorporate the LMS Filter block into your system.

a Connect the output of the Random Source block to the Input port of the LMS Filter block. In the aircraft scenario, the random noise is the white noise measured by the sensor on the outside of the airplane. The LMS Filter block models the effect of the airplane's fuselage on the noise.

b Connect the output of the Digital Filter Design block to the Desired port on the LMS Filter block. This is the signal you want the LMS block to reproduce.

c Connect the output of the LMS Filter block to the negative port of the Sum block you added in step 2.

d Connect the output of the first Sum block to the positive port of the second Sum block. Your model should now look similar to the following figure.



The positive input to the second Sum block is the sum of the input signal and the low frequency noise, $s(n) + y$. The negative input to the second Sum block is the LMS Filter block's best estimation of the low frequency noise,

y'. When you subtract the two signals, you are left with an approximation of the input signal.

$$s(n)_{approx} = s(n) + y - y'$$

In this equation:

- $s(n)$ is the input signal
- $s(n)_{approx}$ is the approximation of the input signal
- y is the noise created by the Random Source block and the Digital Filter Design block
- y' is the LMS Filter block's approximation of the noise

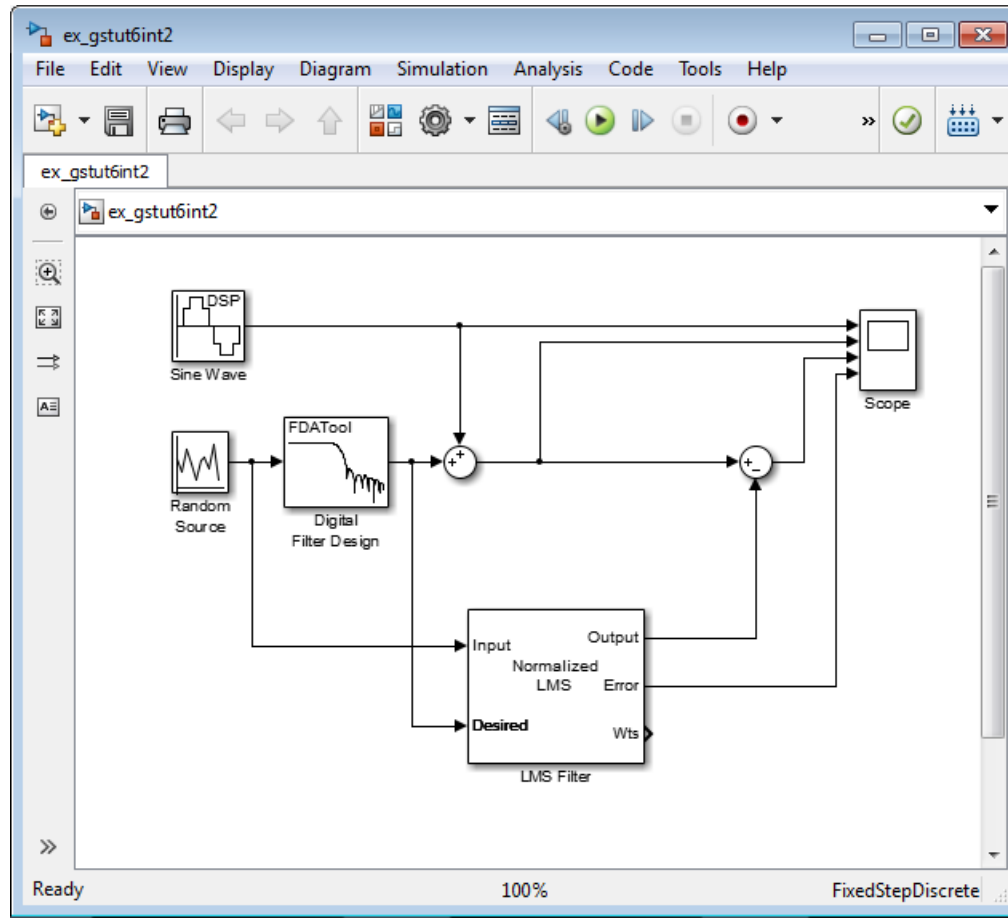
Because the LMS Filter block can only approximate the noise, there is still a difference between the input signal and the approximation of the input signal. In subsequent steps, you set up the Scope block so you can compare the original sinusoidal signal with its approximation.

4 Add two additional inputs and axes to the Scope block. Open the Scope dialog box by double-clicking the Scope block. Click the **Parameters** button. For the **Number of axes** parameter, enter `4`. Close the dialog box by clicking **OK**.

5 Label the new Scope axes. In the Scope window, right-click on the third axes and select **Axes properties**. The Scope properties: axis 3 dialog box opens. In the **Title** box, enter `Approximation of Input Signal`. Close the dialog box by clicking **OK**. Repeat this procedure for the fourth axes and label it `Error`.

6 Connect the output of the second Sum block to the third port of the Scope block.

7 Connect the output of the Error port on the LMS Filter block to the fourth port of the Scope block. Your model should now look similar to the following figure.



In this example, the output of the Error port is the difference between the LMS filter's desired signal and its output signal. Because the error is never zero, the filter continues to modify the filter coefficients in order to better approximate the low frequency noise. The better the approximation, the more low frequency noise that can be removed from the sinusoidal signal. In the next topic, "View the Coefficients of Your Adaptive Filter" on page 3-21, you learn how to view the coefficients of your adaptive filter as they change with time.

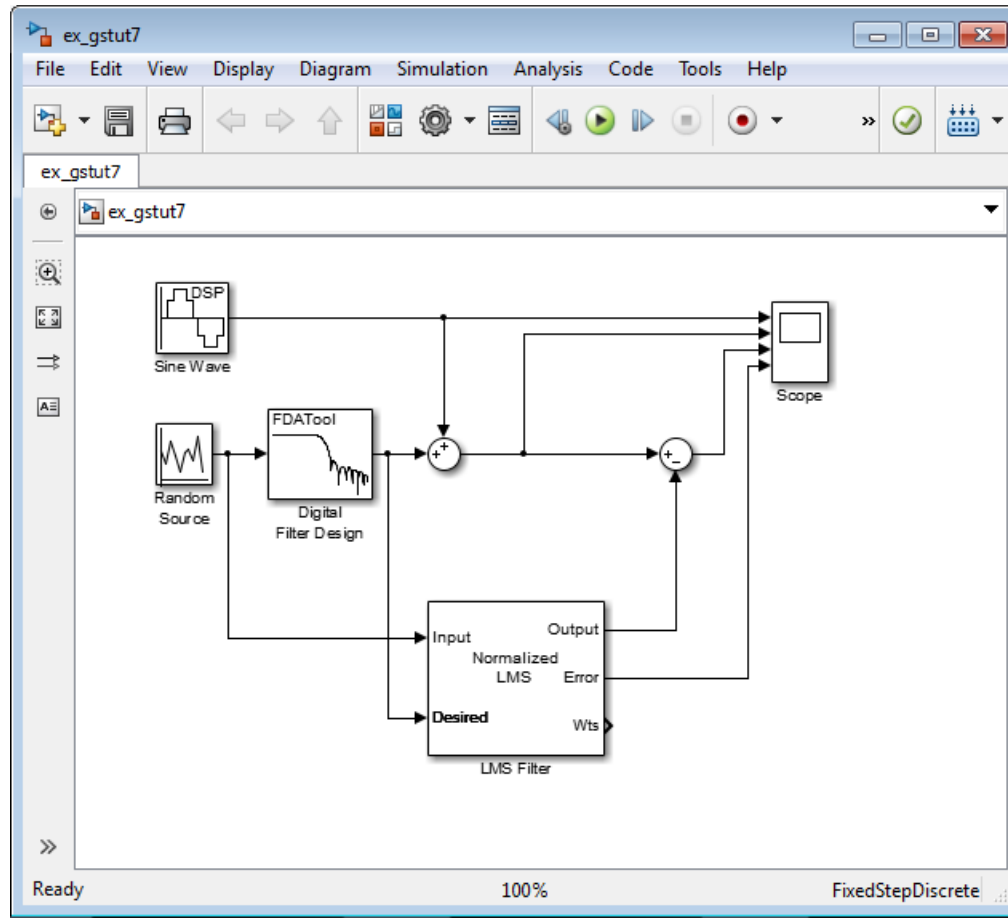
View the Coefficients of Your Adaptive Filter

The coefficients of an adaptive filter change with time in accordance with a chosen algorithm. Once the algorithm optimizes the filter's performance, these filter coefficients reach their steady-state values. You can view the variation of your coefficients, while the simulation is running, to see them settle to their steady-state values. Then, you can determine whether you can implement these values in your actual system:

1. If the model you created in "Add an Adaptive Filter to Your Model" on page 3-16 is not open on your desktop, you can open an equivalent model by typing

   ```
   ex_gstut7
   ```

   at the MATLAB command prompt. Note that the Wts port of the adaptive filter, which outputs the filter weights, still needs to be connected.



2 Open the DSP System Toolbox library by typing `dsplib` at the MATLAB command prompt.

3 View the filter coefficients using a Vector Scope block from the Sinks library.

4 Open the Vector Scope dialog box by double-clicking the block. Set the block parameters as follows:

   a Click the **Scope Properties** tab.

      - **Input domain** = `Time`

- **Time display span (number of frames)** = `1`

b Click the **Display Properties** tab.

- Select the following check boxes:
  - **Show grid**
  - **Frame number**
  - **Compact display**
  - **Open scope at start of simulation**

c Click the **Axis Properties** tab.

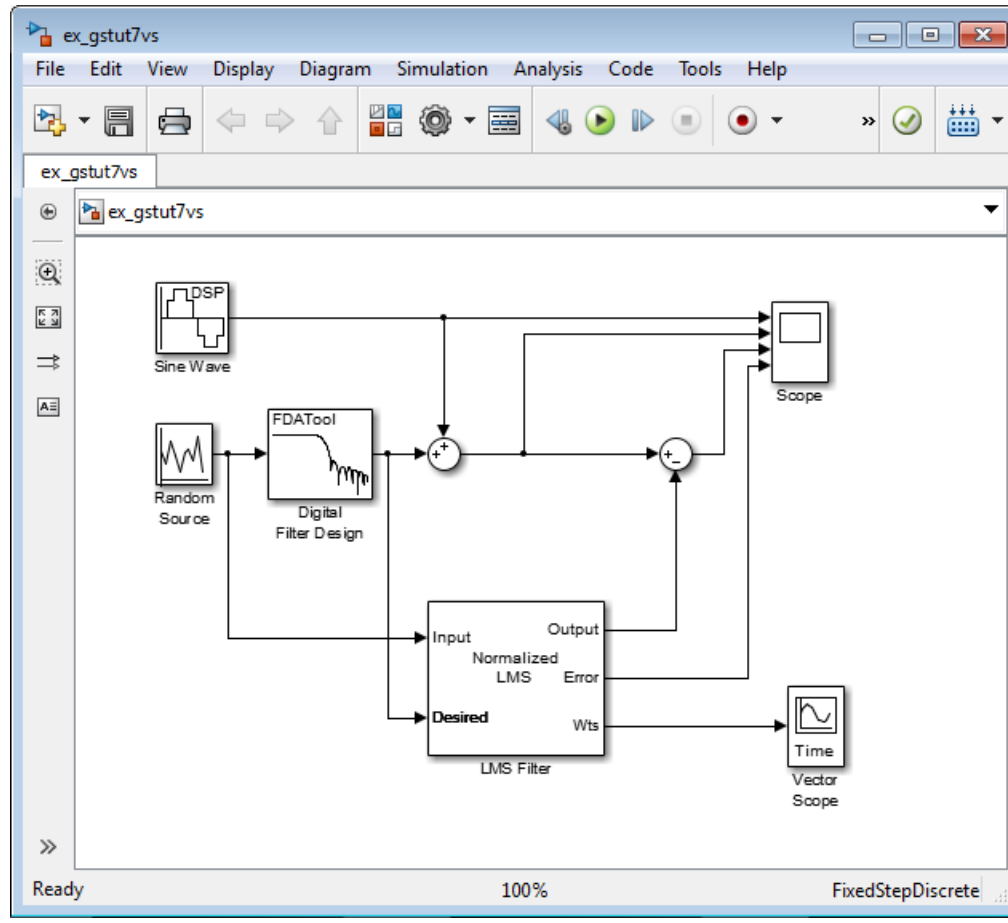
- **Minimum Y-limit** = `-0.2`
- **Maximum Y-limit** = `0.6`
- **Y-axis label** = `Filter Weights`

d Click the **Line Properties** tab.

- **Line visibilities** = `on`
- **Line style** = **:**
- **Line markers** = **.**
- **Line colors** = `[0 0 1]`

e Click **OK**.

5 Connect the Wts port of the LMS Filter block to the Vector Scope block.



6 Set the configuration parameters:

   a Open the Configuration Parameters dialog box by selecting **Model Configuration Parameters** from the **Simulation** menu, and navigate to the **Solver** pane.

   b Enter `inf` for the **Stop time** parameter.

   c Choose `Fixed-step` from the **Type** list.

   d Choose `Discrete (no continuous states)` from the **Solver** list.

We recommend these configuration parameters for models that contain DSP System Toolbox blocks. Because these blocks calculate values directly rather than solving differential equations, you must configure the Simulink Solver to behave like a scheduler. The Solver, while in scheduler mode, uses a block's sample time to determine when the code behind each block is executed. For example, the sample time of the Sine Wave and Random Source blocks in this model is `0.05`. The Solver executes the code behind these blocks, and every other block with this sample time, once every 0.05 second.

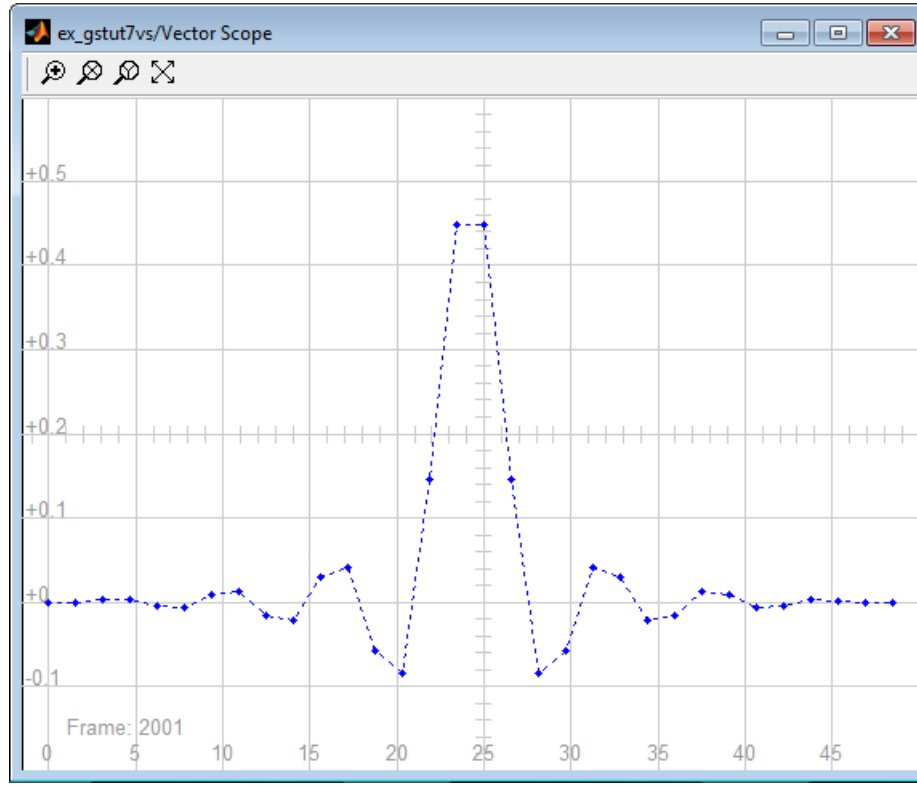
Note When working with models that contain DSP System Toolbox blocks, use source blocks that enable you to specify their sample time. If your source block does not have a **Sample time** parameter, you must add a Zero-Order Hold block in your model and use it to specify the sample time. For more information, see "Continuous-Time Source Blocks" in the *DSP System Toolbox User's Guide*. The exception to this rule is the Constant block, which can have a constant sample time. When it does, Simulink executes this block and records the constant value once, which allows for faster simulations and more compact generated code.

---

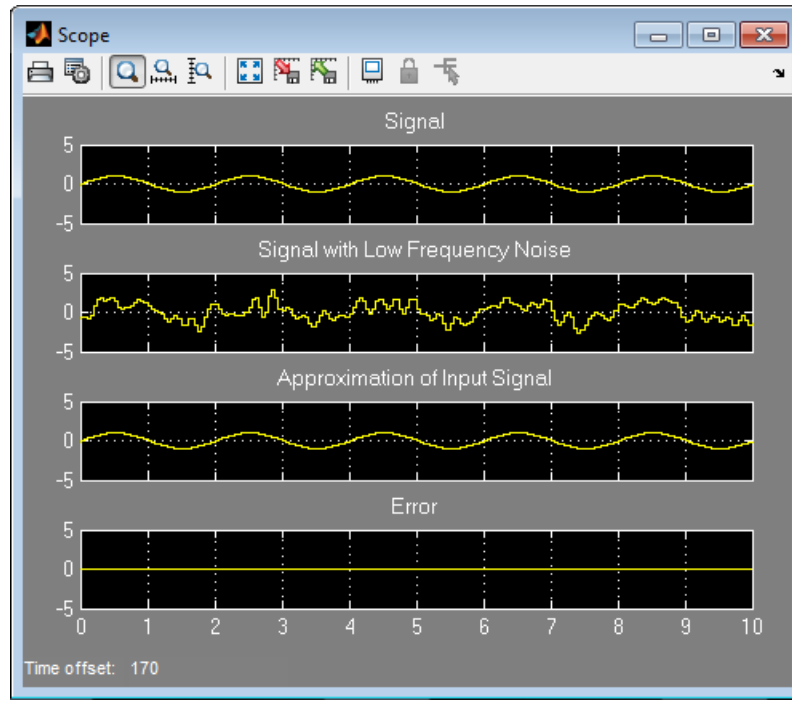
7 Close the dialog box by clicking **OK**.

8 Open the Scope window by double-clicking the Scope block.

9 Run your model and view the behavior of your filter coefficients in the Vector Scope window, which opens automatically when your simulation starts. Over time, you see the filter coefficients change and approach their steady-state values, shown below.



You can simultaneously view the behavior of the system in the Scope window. Over time, you see the error decrease and the approximation of the input signal more closely match the original sinusoidal input signal.



You have now created a model capable of adaptive noise cancellation. So far, you have learned how to design a lowpass filter using the Digital Filter Design block. You also learned how to create an adaptive filter using the LMS Filter block. The DSP System Toolbox product has other blocks capable of designing and implementing digital and adaptive filters. For more information on the filtering capabilities of this product, see “Filter Design” and “Filter Analysis”.

Because all blocks in this model have the same sample time, this model is single rate and Simulink ran it in `SingleTasking` solver mode. If the blocks in your model have different sample times, your model is multirate and Simulink might run it in `MultiTasking` solver mode. For more information on solver modes, see “Recommended Settings for Discrete-Time Simulations” in the *DSP System Toolbox User’s Guide*.

To learn how to generate code from your model using the Simulink Coder product, see the “Generate Code from Simulink” section.

4

System Objects

- "What Is a System Toolbox?" on page 4-2
- "What Are System Objects?" on page 4-3
- "When to Use System Objects Instead of MATLAB Functions" on page 4-5
- "System Design and Simulation in MATLAB" on page 4-8
- "System Design and Simulation in Simulink" on page 4-9
- "System Objects in MATLAB Code Generation" on page 4-10
- "System Objects in Simulink" on page 4-17
- "System Object Methods" on page 4-18
- "System Design in MATLAB Using System Objects" on page 4-21
- "System Design in Simulink Using System Objects" on page 4-28

What Is a System Toolbox?

System Toolbox products provide algorithms and tools for designing, simulating, and deploying dynamic systems in MATLAB and Simulink. These toolboxes contain MATLAB functions, System objects, and Simulink blocks that deliver the same design and verification capabilities across MATLAB and Simulink, enabling more effective collaboration among system designers. Available System Toolbox products include:

- DSP System Toolbox
- Communications System Toolbox
- Computer Vision System Toolbox
- Phased Array System Toolbox

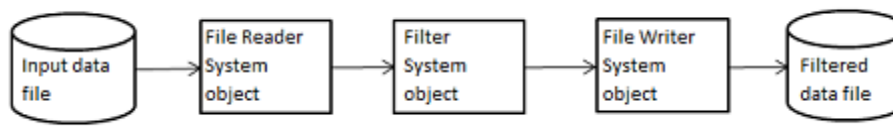
System Toolboxes support floating-point and fixed-point streaming data simulation for both sample- and frame-based data. They provide a programming environment for defining and executing code for various aspects of a system, such as initialization and reset. System Toolboxes also support code generation for a range of system development tasks and workflows, such as:

- Rapid development of reusable IP and test benches
- Sharing of component libraries and systems models across teams
- Large system simulation
- C-code generation for embedded processors
- Finite wordlength effects modeling and optimization
- Ability to prototype and test on real-time hardware

What Are System Objects?

A System object is a specialized kind of MATLAB object. System Toolboxes include System objects and most System Toolboxes also have MATLAB functions and Simulink blocks. System objects are designed specifically for implementing and simulating dynamic systems with inputs that change over time. Many signal processing, communications, and controls systems are dynamic. In a dynamic system, the values of the output signals depend on both the instantaneous values of the input signals and on the past behavior of the system. System objects use internal states to store that past behavior, which is used in the next computational step. As a result, System objects are optimized for iterative computations that process large streams of data, such as video and audio processing systems.

For example, you could use System objects in a system that reads data from a file, filters that data and then writes the filtered output to another file. Typically, a specified amount of data is passed to the filter in each loop iteration. The file reader object uses a state to keep track of where in the file to begin the next data read. Likewise, the file writer object keeps tracks of where it last wrote data to the output file so that data is not overwritten. The filter object maintains its own internal states to assure that the filtering is performed correctly. This diagram represents a single loop of the system.



Many System objects support:

- Fixed-point arithmetic (requires a Fixed-Point Designer™ license)
- C code generation (requires a MATLAB Coder or Simulink Coder license)
- HDL code generation (requires an HDL Coder™ license)
- Executable files or shared libraries generation (requires a MATLAB Compiler™ license)

Note Check your product documentation to confirm fixed-point, code generation, and MATLAB Compiler support for the specific System objects you want to use.

In addition to the System objects provided with System Toolboxes, you can also create your own System objects. See “Define New System Objects”.

When to Use System Objects Instead of MATLAB Functions

In this section...

"System Objects vs. MATLAB Functions" on page 4-5

"Process Audio Data Using Only MATLAB Functions Code" on page 4-5

"Process Audio Data Using System Objects" on page 4-6

System Objects vs. MATLAB Functions

Many System objects have MATLAB function counterparts. For simple, one-time computations use MATLAB functions. However, if you need to design and simulate a system with many components, use System objects. Using System objects is also appropriate if your computations require managing internal states, have inputs that change over time or process large streams of data.

Building a dynamic system with different execution phases and internal states using only MATLAB functions would require complex programming. You would need code to initialize the system, validate data, manage internal states, and reset and terminate the system. System objects perform many of these managerial operations automatically during execution. By combining System objects in a program with other MATLAB functions, your can streamline your code and improve efficiency.

Process Audio Data Using Only MATLAB Functions Code

This example shows how to write MATLAB function-only code for reading audio data.

The code reads audio data from a file, filters it, and then plays the filtered audio data. The audio data is read in frames. This code produces the same result as the System objects code in the next example, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Obtain the total number of samples and the sampling rate from the source file.

```
audioInfo = audioinfo(fname);
maxSamples = audioInfo.TotalSamples;
fs = audioInfo.SampleRate;
```

Define the filter to use.

```
b = fir1(160,.15);
```

Initialize the filter states.

```
z = zeros(1,numel(b)-1);
```

Define the amount of audio data to process at one time, and initialize the while loop index.

```
frameSize = 1024;
nIdx = 1;
```

Define the while loop to process the audio data.

```
while nIdx <= maxSamples(1)-frameSize+1
    audio = audioread(fname,[nIdx nIdx+frameSize-1]);
    [y,z] = filter(b,1,audio,z);
    sound(y,fs);
    nIdx = nIdx+frameSize;
end
```

The loop uses explicit indexing and state management, which can be a tedious and error-prone approach. You must have detailed knowledge of the states, such as, sizes and data types. Another issue with this MATLAB-only code is that the `sound` function is not designed to run in real time. The resulting audio is very choppy and barely audible.

Process Audio Data Using System Objects

This example shows how to write System objects code for reading audio data.

The code uses System objects from the DSP System Toolbox software to read audio data from a file, filter it, and then play the filtered audio data. This code produces the same result as the MATLAB code shown previously, allowing you to compare approaches.

Locate source audio file.

```
fname = 'speech_dft_8kHz.wav';
```

Define the System object to read the file.

```
audioIn = dsp.AudioFileReader(fname,'OutputDataType','single');
```

Define the System object to filter the data.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Define the System object to play the filtered audio data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

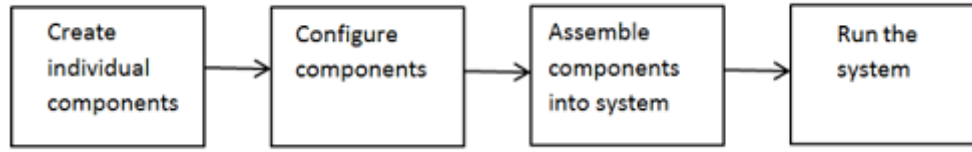
Define the while loop to process the audio data.

```
while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);   % Filter the data
    step(audioOut,y);         % Play the filtered data
end
```

This System objects code avoids the issues present in the MATLAB-only code. Without requiring explicit indexing, the file reader object manages the data frame sizes while the filter manages the states. The audio player object plays each audio frame as it is processed.

System Design and Simulation in MATLAB

System objects allow you to design and simulate your system in MATLAB. You use System objects in MATLAB as shown in this diagram.



1. Create individual components — Create the System objects to use in your system. See "Create Components for Your System" on page 4-21 for information. In addition to the System objects provided with System Toolboxes, you can also create your own System objects. See "Define New System Objects".

2. Configure components — If necessary, change the objects' property values to model your particular system. All System object properties have default values that you may be able to use without changing them. See "Configure Components for Your System" on page 4-22 for information.

3. Assemble components into system — Write a MATLAB program that includes those System objects, connecting them using MATLAB variable as inputs and outputs to simulate your system. See "Assemble Components to Create Your System" on page 4-23 for information.

4. Run the system — Run your program, which uses the `step` method to run your system's System objects. You can change tunable properties while your system is running. See "Run Your System" on page 4-25 and "Reconfigure Your System During Runtime" on page 4-25 for information.

System Design and Simulation in Simulink

You can use System objects in your model to simulate in Simulink.

1. Create a System object to be used in your model. See “Define New Kinds of System Objects for Use in Simulink” on page 4-28 for information.
2. Test your new System object in MATLAB. See “Test New System Objects in MATLAB” on page 4-34
3. Add the System object to your model using the MATLAB System block. See “Add System Objects to Your Simulink Model” on page 4-35 for information.
4. Add other Simulink blocks as needed and connect the blocks to construct your system.
5. Run the system

System Objects in MATLAB Code Generation

In this section...

"System Objects in Generated Code" on page 4-10

"System Objects in codegen" on page 4-16

"System Objects in the MATLAB Function Block" on page 4-16

"System Objects in the MATLAB System Block" on page 4-16

"System Objects and MATLAB® Compiler™ Software" on page 4-16

System Objects in Generated Code

You can generate C/C++ code in MATLAB from your system that contains System objects by using the MATLAB Coder product. Using this product, you can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms. You do not need the MATLAB Coder product to generate code in Simulink.

Note Most, but not all, System objects support code generation. Refer to the particular object's reference page for information.

System Objects Code with Persistent Objects for Code Generation

This example shows how to use System objects to make MATLAB code suitable for code generation. The example highlights key factors to consider, such as passing property values and using extrinsic functions. It also shows that by using persistent objects, the object states are maintained between calls.

```
function w = lmssystem(x, d)
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter
%#codegen

    % Declare System objects as persistent
    persistent hlms;
```

```
    % Initialize persistent System objects only once.
    % Do this with 'if isempty(persistent variable).'
    % This condition will be false after the first time.

    if isempty(hlms)
        % Create LMS adaptive filter used for system
        % identification. Pass property value arguments
        % as constructor arguments. Property values must
        % be constants during compile time.

        hlms = dsp.LMSFilter(11,'StepSize',0.01);
    end

    [~,~,w] = step(hlms,x,d);      % Filter weights
end
```

This example shows how to compile the `lmssystem` function and produce a MEX file with the same name in the current directory.

```
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter

coefs = fir1(10,.25);
hfilt = dsp.FIRFilter('Numerator', coefs);

x = randn(1000,1);                 % Input signal
hSrc = dsp.SignalSource(x,100);    % Use x as input-signal with
                                   % 100 samples per frame

% Generate code for lmssystem
codegen lmssystem -args {ones(100,1),ones(100,1)}

while ~isDone(hSrc)
    in = step(hSrc);
    d  = step(hfilt,in) + 0.01*randn(100,1);  % Desired signal
    w  = lmssystem_mex(in,d);          % Call generated mex file
    stem([coefs.',w]);
end
```

For another detailed code generation example, see "Generate Code for MATLAB Handle Classes and System Objects" in the MATLAB Coder product documentation.

System Objects Code Without Persistent Objects for Code Generation

The following example, using System objects, does not use the persistent keyword because calling a persistent object with different data types causes a data type mismatch error. This example filters the input and then performs a discrete cosine transform on the filtered output. Each call to the `FilterAndDCTLib` function is independent and state information is not retained between calls.

```
function [out] = FilterAndDCTLib(in)
   hFIR = dsp.FIRFilter('Numerator',fir1(10,0.5));
   DCT = dsp.DCT;

   % Run the objects to get the filtered spectrum
   firOut = hFIR.step(in);
   out = hDCT.step(firOut);


function [out1, out2] = CompareRealInt(in1)
   % Call the library function, FilterAndDCTLib, which can
   % generate code for multiple calls each with a different data type.

   % Convert input data from double to int16
   in2 = int16(in1);

   % Call the library function for both data types, double and int16
   out1 = FilterAndDCTLib(in1);
   out2 = FilterAndDCTLib(in2);


function RunDCTExample
   % Execute everything needed at the command line to run the example

   warnState = warning('off','SimulinkFixedPoint:util:fxpParameterUnderflow
```

```
    % Create vector, length 256, of data containing noise and sinusoids
    dataLength = 256;
    sampleData = rand(dataLength,1) + 3*sin(2*pi*[1:dataLength]*.085)' ...
              + 2*cos(2*pi*[1:dataLength]*.02)';

    % Generate code and run generated file
    codegen CompareRealInt -args {sampleData}
    [out1,out2] = CompareRealInt_mex(sampleData);

    % Compare the the floating point results, in blue
    % with the int16 results, in red
    plot(out1,'b');
    hold on;
    plot(out2,'r');
    hold off

    warning(warnState.state,warnState.identifier);
end
```

Usage Rules and Limitations for System Objects in Generated MATLAB Code

The following usage rules and limitations apply to using System objects in code generated from MATLAB.

Object Construction and Initialization

- If objects are stored in persistent variables, initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty( )`.
- Set arguments to System object constructors as compile-time constants.
- You cannot initialize System objects properties with other MATLAB class objects as default values in code generation. You must initialize these properties in the constructor.

Inputs and Outputs

- The data type of the inputs should not change.

- If you want the size of inputs to change, verify that variable-size is enabled. Code generation support for variable-size data also requires that the `Enable variable sizing` option is enabled, which is the default in MATLAB.

Note Variable-size properties in MATLAB Function block in Simulink are not supported. System objects predefined in the software do not support variable-size if their data exceeds the `DynamicMemoryAllocationThreshold` value.

- Do not set System objects to become outputs from the MATLAB Function block.
- Do not use the Save and Restore Simulation State as SimState option for any System object in a MATLAB Function block.
- Do not pass a System object as an example input argument to a function being compiled with `codegen`.
- Do not pass a System object to functions declared as extrinsic (functions called in interpreted mode) using the `coder.extrinsic` function. System objects returned from extrinsic functions and scope System objects that automatically become extrinsic can be used as inputs to another extrinsic function, but do not generate code.

Tunable and Nontunable Properties

- The value assigned to a nontunable property must be a constant and there can be at most one assignment to that property (including the assignment in the constructor).
- For most System objects, the only time you can set their nontunable properties during code generation is when you construct the objects.
  - For System objects that are predefined in the software, you can set their tunable properties at construction time or using dot notation after the object is locked.
  - For System objects that you define, you can change their tunable properties at construction time or using dot notation during code generation.

- Objects cannot be used as default values for properties.
- In MATLAB simulations, default values are shared across all instances of an object. Two instances of a class can access the same default value if that property has not been overwritten by either instance.

Cell Arrays and Global Variables

- Do not use cell arrays.
- Global variables are not supported. To avoid syncing global variables between a MEX file and the workspace, use a coder configuration object. For example:

  ```
  f = coder.MEXConfig;
  f.GlobalSyncMethod='NoSync'
  ```

  Then, include `'-config f'` in your `codegen` command.

Methods

- Code generation support is available only for these System object methods:
  - `get`
  - `getNumInputs`
  - `getNumOutputs`
  - `isDone` (for sources only)
  - `release`
  - `reset`
  - `set` (for tunable properties)
  - `step`

- Code generation support for using dot notation depends on whether the System object is predefined in the software or is one that you defined.
  - For System objects that are predefined in the software, you cannot use dot notation to call methods.

- For System objects that you define, you can use dot notation or function call notation, with the System object as first argument, to call methods.

System Objects in codegen

You can include System objects in MATLAB code in the same way you include any other elements. You can then compile a MEX file from your MATLAB code by using the `codegen` command, which is available if you have a MATLAB Coder license. This compilation process, which involves a number of optimizations, is useful for accelerating simulations. See "Getting Started with MATLAB Coder" and "MATLAB Classes" for more information.

Note Most, but not all, System objects support code generation. Refer to the particular object's reference page for information.

System Objects in the MATLAB Function Block

Using the MATLAB Function block, you can include any System object and any MATLAB language function in a Simulink model. This model can then generate embeddable code. System objects provide higher-level algorithms for code generation than do most associated blocks. For more information, see "What Is a MATLAB Function Block?" in the Simulink documentation.

System Objects in the MATLAB System Block

Using the MATLAB System block, you can include in a Simulink model individual System objects that you create with a class definition file . The model can then generate embeddable code. For more information, see "What Is the MATLAB System Block?" in the Simulink documentation.

System Objects and MATLAB Compiler Software

MATLAB Compiler software supports System objects for use inside MATLAB functions. The compiler product does not support System objects for use in MATLAB scripts.

System Objects in Simulink

In this section...
"System Objects in the MATLAB Function Block" on page 4-17
"System Objects in the MATLAB System Block" on page 4-17

System Objects in the MATLAB Function Block

You can include System object code in Simulink models using the MATLAB Function block. Your function can include one or more System objects. Portions of your system may be easier to implement in the MATLAB environment than directly in Simulink. Many System objects have Simulink block counterparts with equivalent functionality. Before writing MATLAB code to include in a Simulink model, check for existing blocks that perform the desired operation.

System Objects in the MATLAB System Block

You can include individual System objects that you create with a class definition file into Simulink using the MATLAB System block. This provides one way to add your own algorithm blocks into your Simulink models. For information, see "System Object Integration" in the Simulink documentation.

System Object Methods

In this section...

"What Are System Object Methods?" on page 4-18

"The Step Method" on page 4-18

"Common Methods" on page 4-19

What Are System Object Methods?

After you create a System object, you use various object methods to process data or obtain information from or about the object. All methods that are applicable to an object are described in the reference pages for that object. System object method names begin with a lowercase letter and class and property names begin with an uppercase letter. The syntax for using methods is `<method>(<handle>)`, such as `step(H)`, plus possible extra input arguments.

System objects use a minimum of two commands to process data—a constructor to create the object and the step method to run data through the object. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings. Using this approach avoids repeated input validation and verification, allows for easy use within a programming loop, and improves overall performance. In contrast, MATLAB functions must validate parameters every time you call the function.

These advantages make System objects particularly well suited for processing streaming data, where segments of a continuous data stream are processed iteratively. This ability to process streaming data provides the advantage of not having to hold large amounts of data in memory. Use of streaming data also allows you to use simplified programs that use loops efficiently.

The Step Method

The `step` method is the key System object method. You use `step` to process data using the algorithm defined by that object. The `step` method performs other important tasks related to data processing, such as initialization and handling object states. Every System object has its own customized `step` method, which is described in detail on the `step` reference page for that object.

For more information about the step method and other available methods, see the descriptions in “Common Methods” on page 4-19.

Common Methods

All System objects support the following methods, each of which is described in a method reference page associated with the particular object. In cases where a method is not applicable to a particular object, calling that method has no effect on the object.

Method	Description
`step`	Processes data using the algorithm defined by the object. As part of this processing, it initializes needed resources, returns outputs, and updates the object states. After you call the `step` method, you cannot change any input specifications (i.e., dimensions, data type, complexity). During execution, you can change only tunable properties. The `step` method returns regular MATLAB variables. Example: `Y = step(H,X)`
`release`	Releases any special resources allocated by the object, such as file handles and device drivers, and unlocks the object. For System objects, use the `release` method instead of a destructor.
`reset`	Resets the internal states of the object to the initial values for that object
`getNumInputs`	Returns the number of inputs (excluding the object itself) expected by the `step` method. This number varies for an object depending on whether any properties enable additional inputs.
`getNumOutputs`	Returns the number of outputs expected from the `step` method. This number varies for an object depending on whether any properties enable additional outputs.

Method	Description
`getDiscreteState`	Returns the discrete states of the object in a structure. If the object is unlocked (when the object is first created and before you have run the `step` method on it or after you have released the object), the states are empty. If the object has no discrete states, `getDiscreteState` returns an empty structure.
`clone`	Creates another object of the same type with the same property values
`isLocked`	Returns a logical value indicating whether the object is locked.
`isDone`	Applies to source objects only. Returns a logical value indicating whether the step method has reached the end of the data file. If a particular object does not have end-of-data capability, this method value returns `false`.
`info`	Returns a structure containing characteristic information about the object. The fields of this structure vary depending on the object. If a particular object does not have characteristic information, the structure is empty.

System Design in MATLAB Using System Objects

In this section...

"Create Components for Your System" on page 4-21

"Configure Components for Your System" on page 4-22

"Assemble Components to Create Your System" on page 4-23

"Run Your System" on page 4-25

"Reconfigure Your System During Runtime" on page 4-25

Create Components for Your System

This example shows how to create components for a system that processes a long stream of audio data. The data is read from a file, filtered, and then played.

A System object is a component you can use to create your system in MATLAB. System objects support fixed- or variable-size data. *Variable-size data* is data whose size can change at run time. By contrast, fixed-size data is data whose size is known and locked at initialization time, and therefore, cannot change at run time.

Many System objects are predefined in the software. You can also create your own System objects (see "Define New System Objects").

The particular predefined components you need are:

- `dsp.AudioFileReader` — Read the file of audio data
- `dsp.FIRFilter` — Filter the audio data
- `dsp.AudioPlayer` — Play the filtered audio data

First, you create the component objects, using default property settings:

```
audioIn = dsp.AudioFileReader;
filtLP = dsp.FIRFilter;
audioOut = dsp.AudioPlayer;
```

Next, you configure each System object for your system. See “Configure Components for Your System” on page 4-22. Alternately, if desired, you can “Create and Configure Components at the Same Time” on page 4-23.

Configure Components for Your System

When to Configure Components

If you did not set an object’s properties when you created it and do not want to use default values, you must explicitly set those properties. Some properties allow you to change their values while your system is running. See “Reconfigure Your System During Runtime” on page 4-25 for information.

Most properties are independent of each other. However, some System object properties enable or disable another property or limit the values of another property. To avoid errors or warnings, you should set the controlling property before setting the dependent property.

Display Component Property Values

To display the current property values for an object, type that object’s handle name at the command line (such as `audioIn`). To display the value of a specific property, type `objecthandle.propertyname` (such as `audioIn.FileName`).

Configure Component Property Values

This example shows how to configure the components for your system by setting the component objects’ properties.

Use this procedure if you have created your components as described in “Create Components for Your System” on page 4-21. If you have not yet created your components, use the procedure in “Create and Configure Components at the Same Time” on page 4-23

For the file reader object, specify the file to read and set the output data type.

```
audioIn.Filename = 'speech_dft_8kHz.wav';
audioIn.OutputDataType = 'single';
```

For the filter object, specify the filter numerator coefficients using the `fir1` function, which specifies the lowpass filter order and the cutoff frequency.

```
filtLP.Numerator = fir1(160,.15);
```

For the audio player object, specify the sample rate. In this case, use the same sample rate as the input data.

```
audioOut.SampleRate = audioIn.SampleRate;
```

Create and Configure Components at the Same Time

This example shows how to create your System object components and configure the desired properties at the same time. To avoid errors or warnings for dependent properties, you should set the controlling property before setting the dependent property. Use this procedure if you have not already created your components.

Create the file reader object, specify the file to read, and set the output data type.

```
audioIn = dsp.AudioFileReader('speech_dft_8kHz.wav',...
    'OutputDataType','single')
```

Create the filter object and specify the filter numerator using the `fir1` function. Specify the lowpass filter order and the cutoff frequency of the `fir1` function.

```
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
```

Create the audio player object and specify the sample rate. In this case, use the same sample rate as the input data.

```
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);
```

After you create the components, you can assemble them in your system. See "Assemble Components to Create Your System" on page 4-23.

Assemble Components to Create Your System

- "Connect Inputs and Outputs" on page 4-24

• "Code for the Whole System" on page 4-24

Connect Inputs and Outputs

After you have determined the components you need and have created and configured your System objects, assemble your system. You use the System objects like other MATLAB variables and include them in MATLAB code. You can pass MATLAB variables into and out of System objects.

The main difference between using System objects and using functions is the `step` method. The `step` method is the processing command for each System object and is customized for that specific System object. This method initializes your objects and controls data flow and state management of your system. You typically use `step` within a loop.

You use the output from an object's `step` method as the input to another object's `step` method. For some System objects, you can use properties of those objects to change the number of inputs or outputs. To verify that the appropriate number of input and outputs are being used, you can use `getNumInputs` and `getNumOutputs` on any System object. For information on all available System object methods, see "System Object Methods" on page 4-18.

Code for the Whole System

This example shows how to write the full code for reading, filtering, and playing a file of audio data.

You can type this code on the command line or put it into a program file.

```
audioIn = dsp.AudioFileReader('speech_dft_8kHz.wav',...
   'OutputDataType','single');
filtLP = dsp.FIRFilter('Numerator',fir1(160,.15));
audioOut = dsp.AudioPlayer('SampleRate',audioIn.SampleRate);

while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);   % Filter the data
    step(audioOut,y);         % Play the filtered data
end
```

The while loop uses the `isDone` method to read through the entire file. The `step` method is used on each object inside the loop.

Now, you are ready to run your system. See "Run Your System" on page 4-25.

Run Your System

- "How to Run Your System" on page 4-25
- "What You Cannot Change While Your System Is Running" on page 4-25

How to Run Your System

Run your code either by typing directly at the command line or running a file containing your program. When you run the code for your system, the `step` method instructs each object to process data through that object.

What You Cannot Change While Your System Is Running

The first call to the `step` method initializes and then locks your object. When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. Use the `isLocked` method to verify whether an object is locked. When the object is locked, you cannot change:

- Number of inputs or outputs
- Data type of inputs or outputs
- Data type of any tunable property
- Dimensions of inputs or tunable properties, except for System objects that support variable-size data
- Value of any nontunable property

To make changes to your system while it is running, see "Reconfigure Your System During Runtime" on page 4-25.

Reconfigure Your System During Runtime

- "When Can You Change Component Properties?" on page 4-26

- "Change a Tunable Property in Your System" on page 4-26
- "Change Input Complexity or Dimensions" on page 4-26

When Can You Change Component Properties?

When a System object has started processing data, it is locked to prevent changes that would disrupt its processing. You can use `isLocked` on any System object to verify whether it is locked or not. When processing is complete, you can use the `release` method to unlock a System object.

Some object properties are *tunable*, which enables you to change them even if the object is locked. Unless otherwise specified, System objects properties are nontunable. Refer to the object's reference page to determine whether an individual property is tunable. Typically, tunable properties are not critical to how the System object processes data.

Change a Tunable Property in Your System

This example shows how to change a tunable property.

You can change the filter type to a high-pass filter as your code is running by replacing the while loop with the following while loop. The change takes effect the next time the `step` method is called (such as at the next iteration of the while loop).

```
reset(audioIn);               % Reset audio file
filtLP.Numerator = fir1(160,0.15,'high');
while ~isDone(audioIn)
    audio = step(audioIn);    % Read audio source file
    y = step(filtLP,audio);   % Filter the data
    step(audioOut,y);         % Play the filtered data
end
```

Change Input Complexity or Dimensions

During simulation, some System objects do not allow complex data if the object was initialized with real data. You cannot change any input complexity during code generation.

You can change the value of a tunable property without a warning or error being produced. For all other changes at run time, an error occurs.

System Design in Simulink Using System Objects

In this section...

"Define New Kinds of System Objects for Use in Simulink" on page 4-28

"Test New System Objects in MATLAB" on page 4-34

"Add System Objects to Your Simulink Model" on page 4-35

Define New Kinds of System Objects for Use in Simulink

This example shows the general steps to create a System object for use in Simulink. The example performs system identification using a least mean squares (LMS) adaptive filter and is similar to the System Identification Using MATLAB System Blocks Simulink example.

A System object is a component you can use to create your system in MATLAB. You can write the code in MATLAB and use that code to create a block in Simulink. To define your own System object, you write a class definition file, which is a text-based MATLAB file that contains the code defining your object. See "System Object Integration" in the Simulink documentation. The LMS Adaptive Filter and Integer Delay blocks in this example model are each from a System object class definition file. These files are described below.

Define System Object with Block Customizations

1 Create a class definition text file to define your System object. This example creates a least mean squares (LMS) filter and includes customizations to the block icon and dialog appearance.

Note Instead of manually creating your class definition file, you can use the **New > System Object > Simulink Extension** menu option to open a template. This template includes customizations of the System object for use in the Simulink MATLAB System block. You edit the template file, using it as guideline, to create your own System object.

---

2. On the first line of the class definition file, specify the name of your System object and subclass from both `matlab.System` and `matlab.system.mixin.CustomIcon`. The `matlab.System` base class enables you to use all the basic System object methods and specify the block input and output names, title, and property groups. The `CustomIcon` mixin class enables the method that lets you specify the block icon.

3. Add the appropriate basic System object methods to set up, reset, set the number of inputs and outputs, and run your algorithm. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.

   - Use the `setupImpl` method to perform one-time calculations and initialize variables.
   - Use the `stepImpl` method to implement the block's algorithm.
   - Use the `resetImpl` to reset the state properties or `DiscreteState` properties.
   - Use the `getNumInputsImpl` and `getNumOutputsImpl` methods to specify the number of inputs and outputs, respectively.

4. Add the appropriate `CustomIcon` methods to define the appearance of the MATLAB System block in Simulink. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.

   - Use the `getHeaderImpl` method to specify the title and description to display on the block dialog.
   - Use the `getPropertyGroupsImpl` method to specify groups of properties to display on the block dialog.
   - Use the `getIconImpl` method to specify the text to display on the block icon.
   - Use the `getInputNamesImpl` and `getOutputNamesImpl` methods to specify the labels to display for the block input and output ports.

   The full class definition file for the least mean squares filter is:

```
classdef lmsSysObj < matlab.System &...
      matlab.system.mixin.CustomIcon
```

```
%lmsSysObj Least mean squares (LMS) adaptive filtering.
%#codegen

properties
   % Mu Step size
   Mu = 0.005;
end

properties (Nontunable)
   % Weights  Filter weights
   Weights = 0;
   % N  Number of filter weights
   N = 32;
end

properties (DiscreteState)
   X;
   H;
end

methods(Access=protected)
   function setupImpl(obj, ~, ~)
      obj.X = zeros(obj.N,1);
      obj.H = zeros(obj.N,1);
   end

   function [y, e_norm] = stepImpl(obj,d,u)
      tmp = obj.X(1:obj.N-1);
      obj.X(2:obj.N,1) = tmp;
      obj.X(1,1) = u;
      y = obj.X'*obj.H;
      e = d-y;
      obj.H = obj.H + obj.Mu*e*obj.X;
      e_norm = norm(obj.Weights'-obj.H);
   end

   function resetImpl(obj)
      obj.X = zeros(obj.N,1);
      obj.H = zeros(obj.N,1);
   end
```

```
        function num = getNumInputsImpl(~)
      num = 2;
   end
   function num = getNumOutputsImpl(~)
      num = 2;
   end
end

% Block icon and dialog customizations
methods(Static, Access=protected)
   function header = getHeaderImpl
      header = matlab.system.display.Header(...
           'lmsSysObj', ...
           'Title', 'LMS Adaptive Filter');
   end

   function groups = getPropertyGroupsImpl
      upperGroup = matlab.system.display.SectionGroup(...
           'Title','General',...
           'PropertyList',{'Mu'}); %#ok<*EMCA>

      lowerGroup = matlab.system.display.SectionGroup(...
           'Title','Coefficients', ...
           'PropertyList',{'Weights','N'}); %#ok<*EMCA>

      groups = [upperGroup,lowerGroup];
   end
end

methods(Access=protected)
   function icon = getIconImpl(~)
      icon = sprintf('LMS Adaptive\nFilter');
   end
   function [in1name, in2name] = getInputNamesImpl(~)
      in1name = 'Desired';
      in2name = 'Actual';
   end
   function [out1name, out2name] = getOutputNamesImpl(~)
      out1name = 'Output';
```

```
            out2name = 'EstError';
        end
    end
end
```

Define System Object with Nondirect Feedthrough

1. Create a class definition text file to define your System object. This example creates an integer delay and includes customizations to the block icon. It implements a System object that you can use for nondirect feedthrough. See“Use System Objects in Feedback Loops” for more information.

2. On the first line of the class definition file, subclass from `matlab.System`, `matlab.system.mixin.CustomIcon`, and `matlab.system.mixin.Nondirect`. The `matlab.System` base class enables you to use all the basic System object methods and specify the block input and output names, title, and property groups. The `CustomIcon` mixin class enables the method that lets you specify the block icon. The `Nondirect` mixin enables the methods that let you specify how the block is updated and what it outputs.

3. Add the appropriate basic System object methods to set up and reset the object and set and validate the properties. Since this object supports nondirect feedthrough, you do not implement the `stepImpl` method. You implement the `updateImpl` and `outputImpl` methods instead. See the reference pages for each method and the full class definition file below for the implementation of each of these methods.

   - Use the `setupImpl` method to initialize some of the object's properties.
   - Use the `resetImpl` to reset the property states.
   - Use the `validatePropertiesImpl` to check that the property values are valid.

4. Add the following `Nondirect` mixin class methods instead of the `stepImpl` method to specify how the block updates its state and its output. See the reference pages and the full class definition file below for the implementation of each of these methods.

   - Use the `outputImpl` method to implement code to calculate the block output.

- Use the `updateImpl` method to implement code to update the block's internal states.
- Use the `isInputDirectFeedthroughImpl` to specify that the block is not direct feedthrough. Its inputs do not directly affect its outputs.

5 Add the `getIconImpl` method to define the block icon when it is used in Simulink via the MATLAB System block. See the reference page and the full class definition file below for the implementation of this method.

The full class definition file for the delay is:

```
classdef intDelaySysObj < matlab.System &...
     matlab.system.mixin.Nondirect &...
     matlab.system.mixin.CustomIcon
   %intDelaySysObj Delay input by specified number of samples.
   %#codegen

   properties
      %InitialOutput Initial output
      InitialOutput = 0;
   end

   properties (Nontunable)
      % NumDelays Number of delays
      NumDelays = 1;
   end

   properties(DiscreteState)
      PreviousInput;
   end

   methods(Access=protected)
      function setupImpl(obj, ~)
         obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
      end

      function [y] = outputImpl(obj, ~)
         % Output does not directly depend on input
         y = obj.PreviousInput(end);
      end
```

```
        function updateImpl(obj, u)
           obj.PreviousInput = [u obj.PreviousInput(1:end-1)];
        end

        function flag = isInputDirectFeedthroughImpl(~,~)
           flag = false;
        end

        function validatePropertiesImpl(obj)
           if ((numel(obj.NumDelays)>1) || (obj.NumDelays <= 0))
              error('Number of delays must be positive non-zero scalar value.
           end
           if (numel(obj.InitialOutput)>1)
              error('Initial output must be scalar value.');
           end
        end

        function resetImpl(obj)
           obj.PreviousInput = ones(1,obj.NumDelays)*obj.InitialOutput;
        end

        function icon = getIconImpl(~)
           icon = sprintf('Integer\nDelay');
        end
     end
  end
```

Test New System Objects in MATLAB

1. Create an instance of your new System object. For example, create an instance of the `lmsSysObj`.

   ```
   s = lmsSysObj;
   ```

2. Run the step method on the object multiple times with different inputs. This tests for syntax errors and other possible issues before you add it to Simulink. For example,

   ```
   desired = 0;
   actual = 0.2;
   ```

```
step(s,desired,actual);
```

Add System Objects to Your Simulink Model

1. Add your System objects to your Simulink model by using the MATLAB System block as described in "Mapping System Objects to Block Dialog Box".

2. Add other Simulink blocks, connect them, and configure any needed parameters to complete your model as described in the Simulink documentation. See the System Identification for an FIR System Using MATLAB System Blocks Simulink example.

3. Run your model in the same way you run any Simulink model.

.