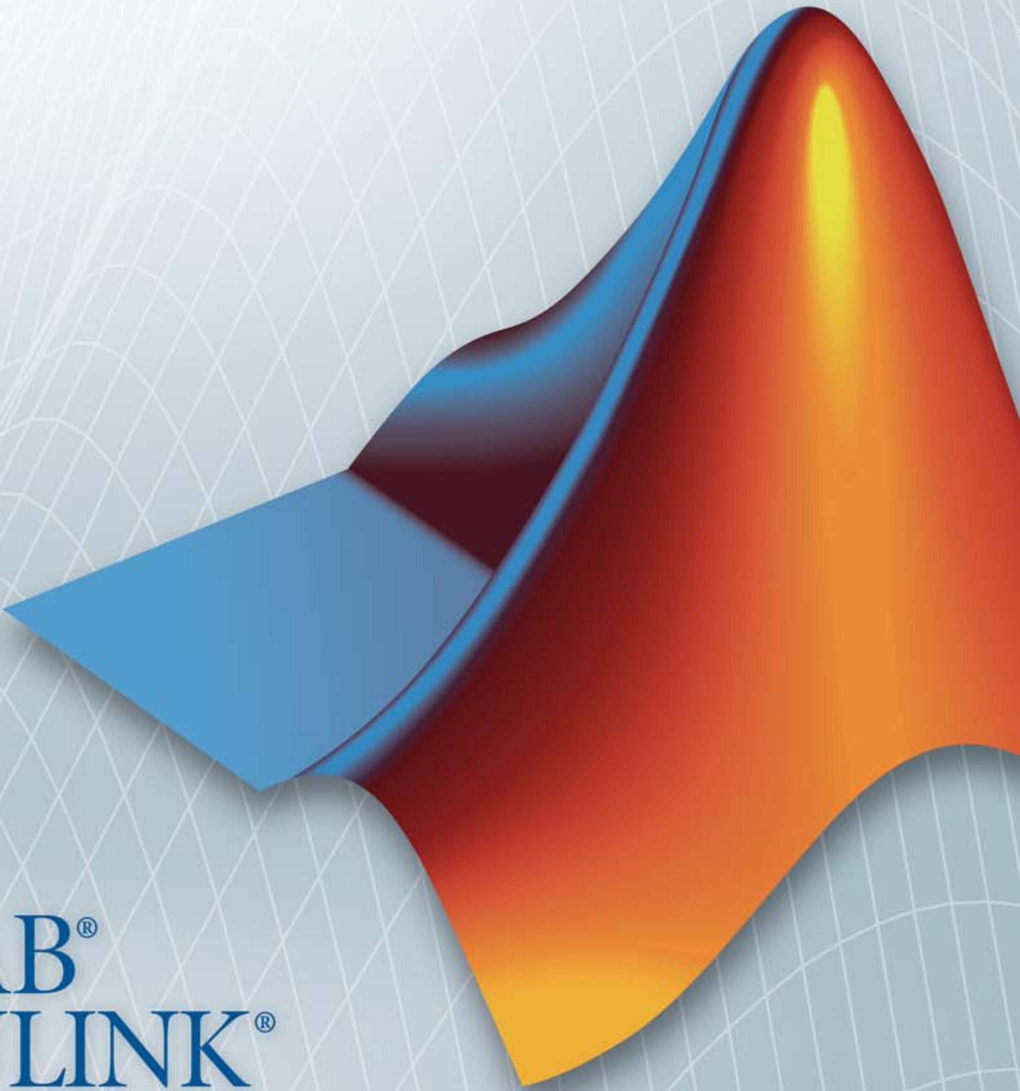
DSP System Toolbox™

Getting Started Guide

R2012b



MATLAB®
&SIMULINK®

MathWorks®

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)

Contents

Introduction

Product Description

Design and simulate signal processing systems

DSP System Toolbox™ provides algorithms and tools for the design and simulation of 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

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

- "Using dspstartup.m" on page 1-4
- "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.

Design a Filter with fdesign and filterbuilder

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.

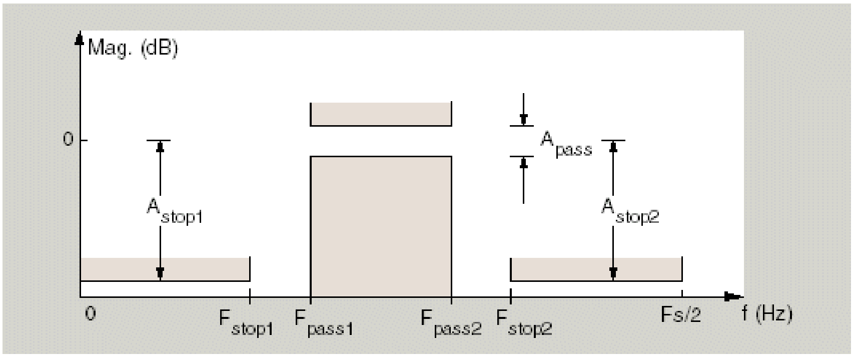
Design a Filter Using fdesign

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

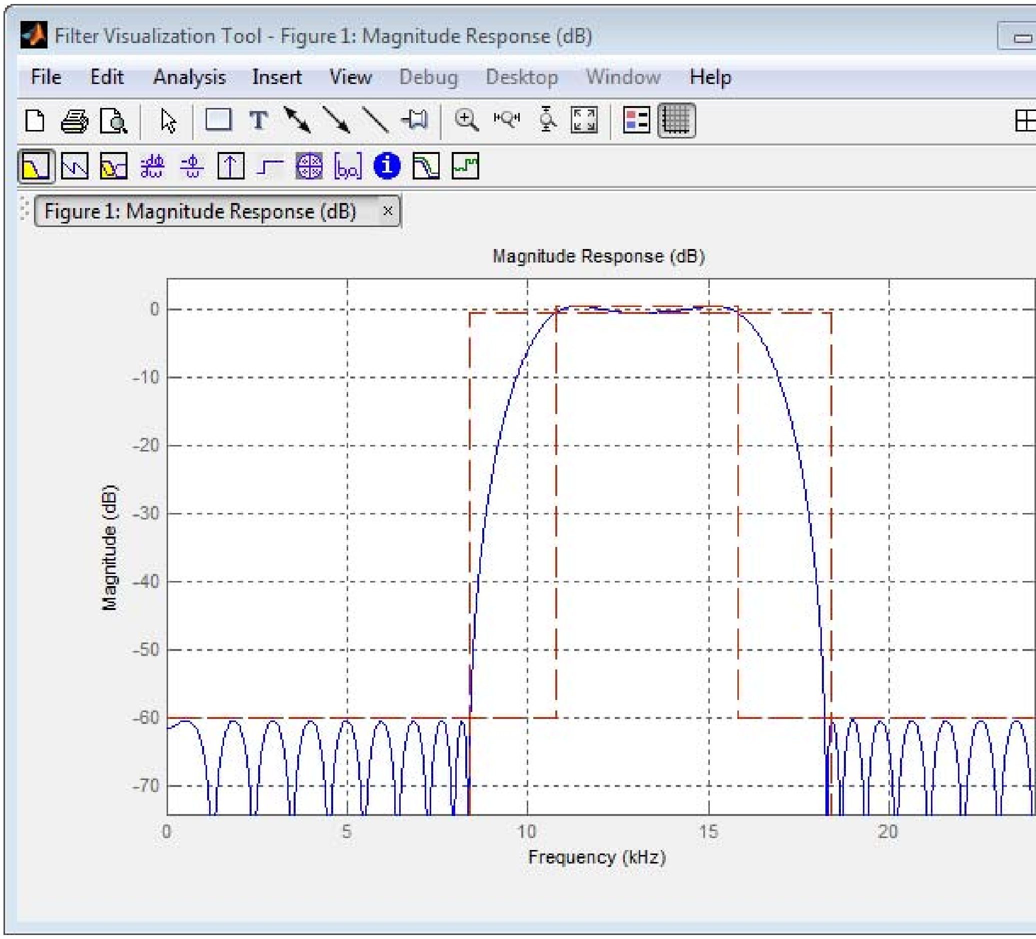
```
>> BandPassFilt = design(BandPassSpecObj)

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.

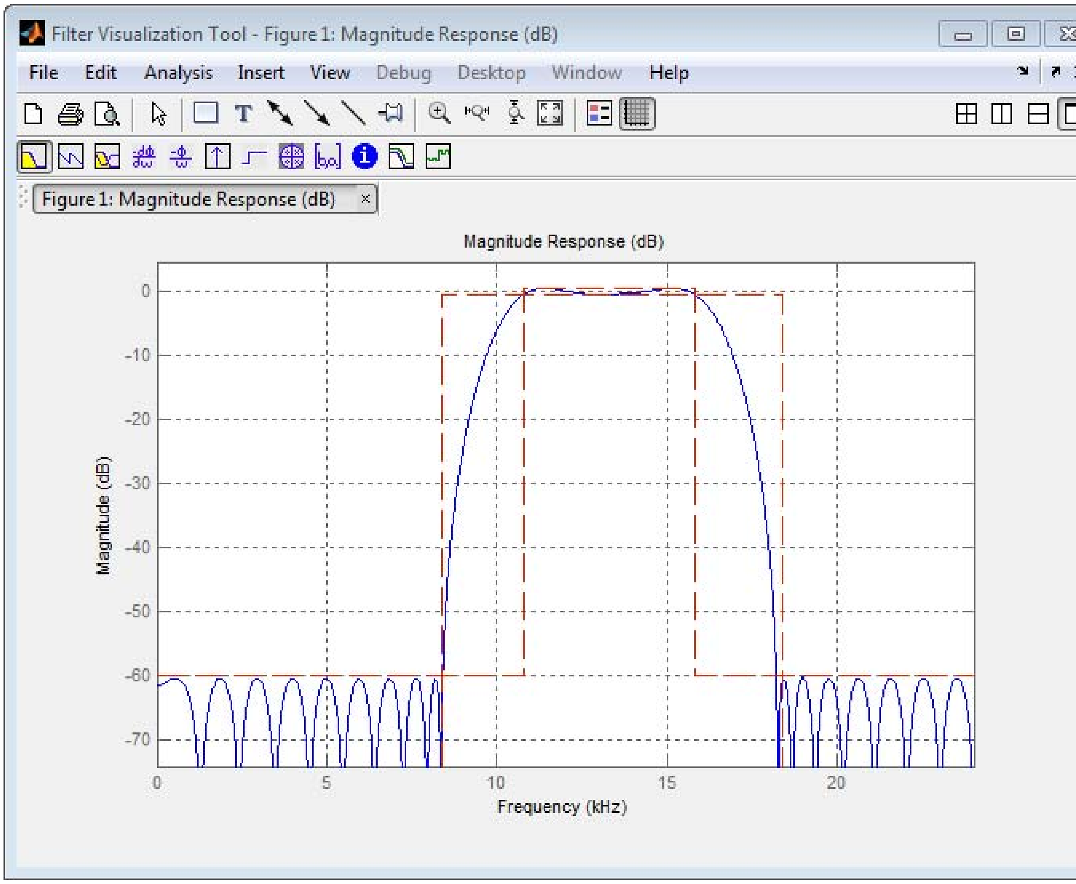
3. Enter the correct frequencies for **Fpass2** and **Fstop2**, then click **OK**. Here the specification uses normalized frequency, so that the passband and stopband edges are expressed as a fraction of the Nyquist frequency (in this case, 48/2 kHz). The following message appears at the MATLAB prompt:

   ```
   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
   ```


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)

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

Design Filters in Simulink

Design and Implement a Filter

In this section...

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.

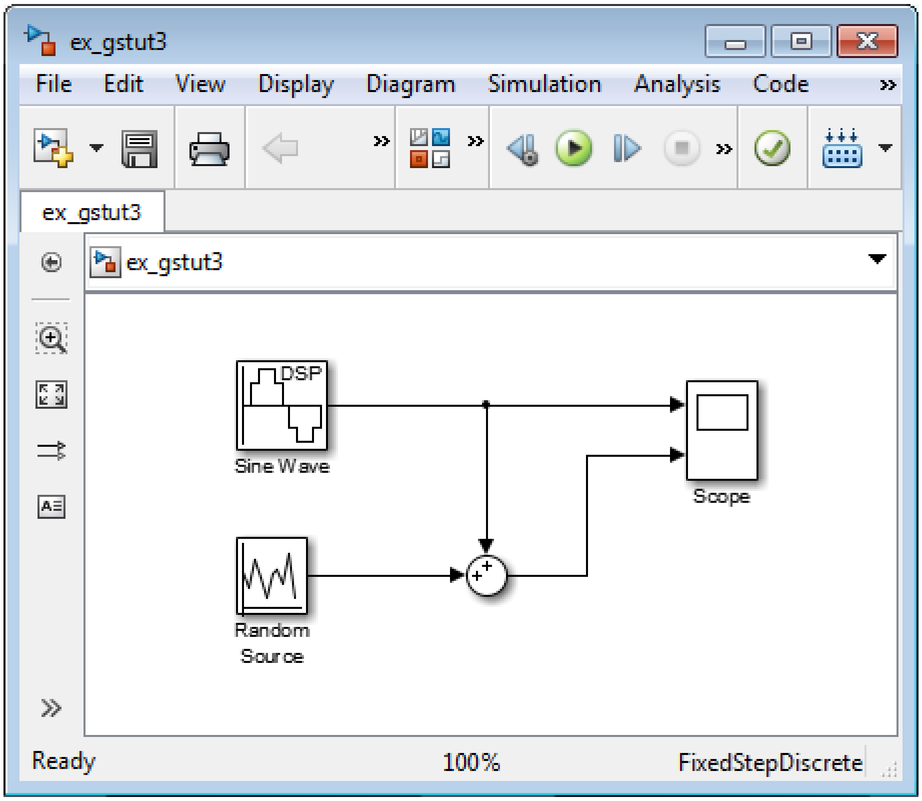
As a practical application, suppose a pilot is speaking into a microphone within the cockpit of an airplane. The noise of the wind passing over the fuselage is also reaching the microphone. A sensor is measuring the noise of the wind on the outside of the plane. You want to estimate the wind noise inside 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
   ```

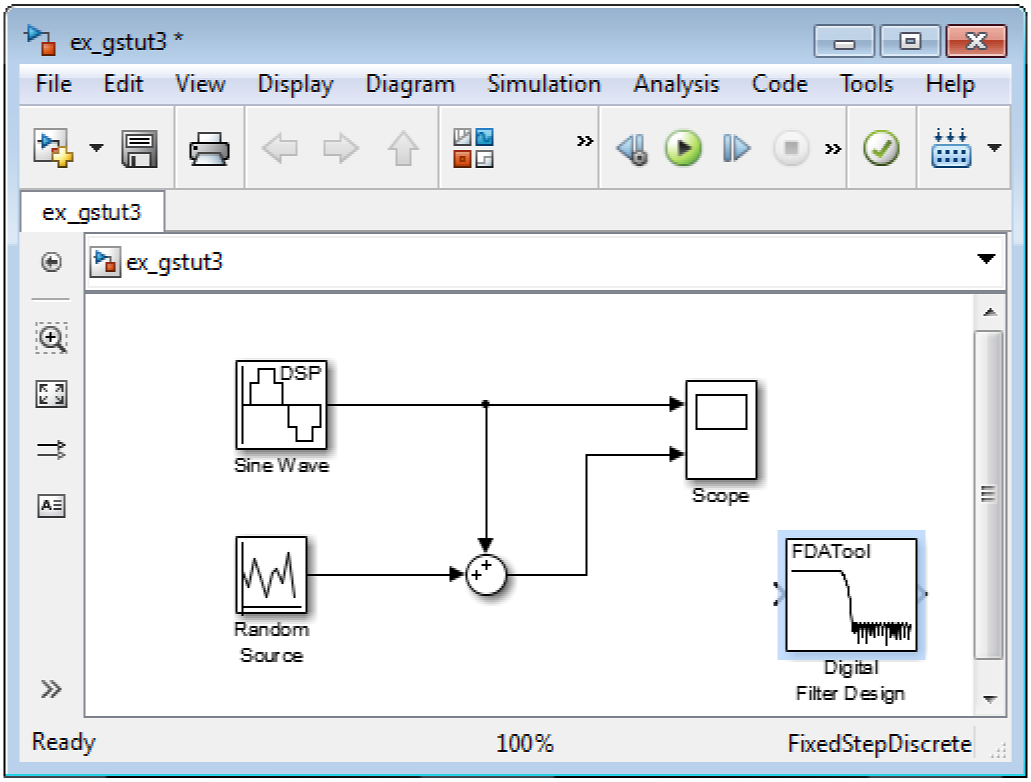
   at the MATLAB command prompt. This model contains a Scope block that displays the original sine wave and the sine wave with white noise added.



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.



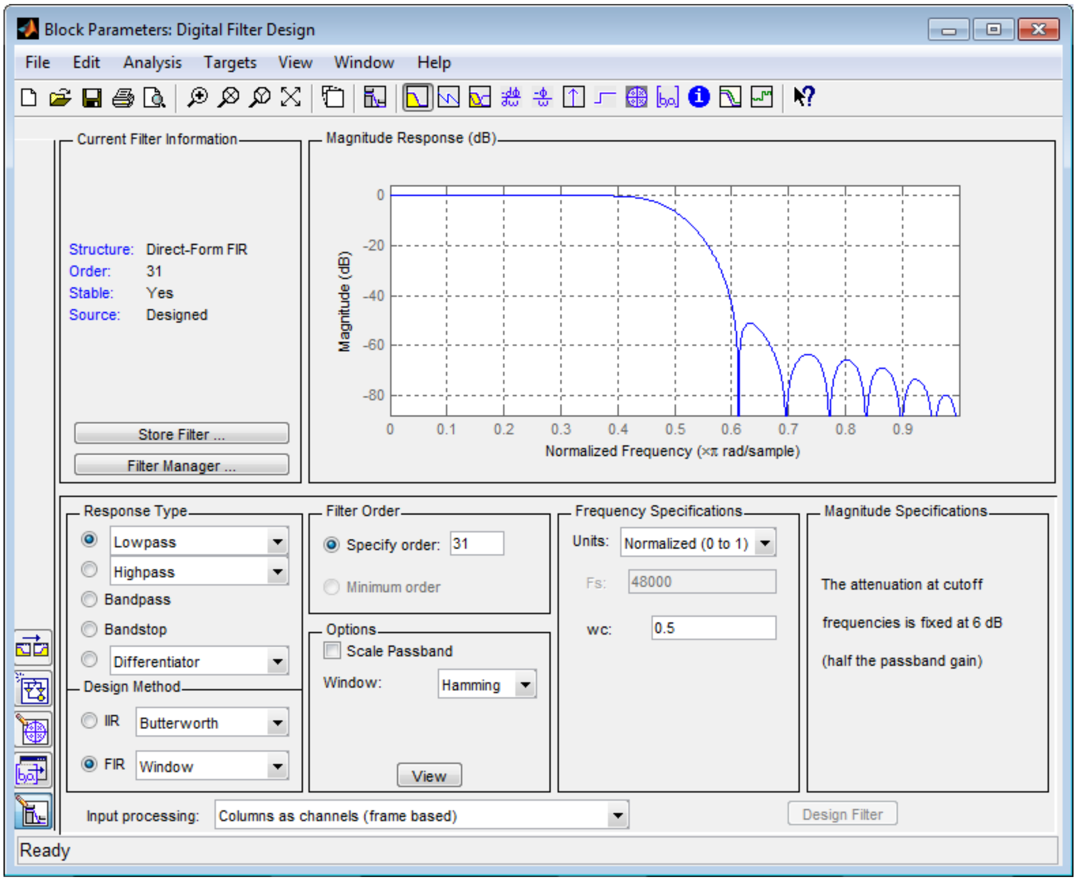
4 Set the Digital Filter Design block parameters to design a lowpass filter and create low frequency noise. Open the block parameters dialog box by double-clicking the block. Set the parameters as follows:

- **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.



Structure: Direct-Form FIR
Order: 31
Stable: Yes
Source: Designed

Response Type: Lowpass; Design Method: FIR Window

Specify order: 31

Window: Hamming

Units: Normalized (0 to 1)

wc: 0.5

The attenuation at cutoff frequencies is fixed at 6 dB (half the passband gain)

Input processing: Columns as channels (frame based)

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.

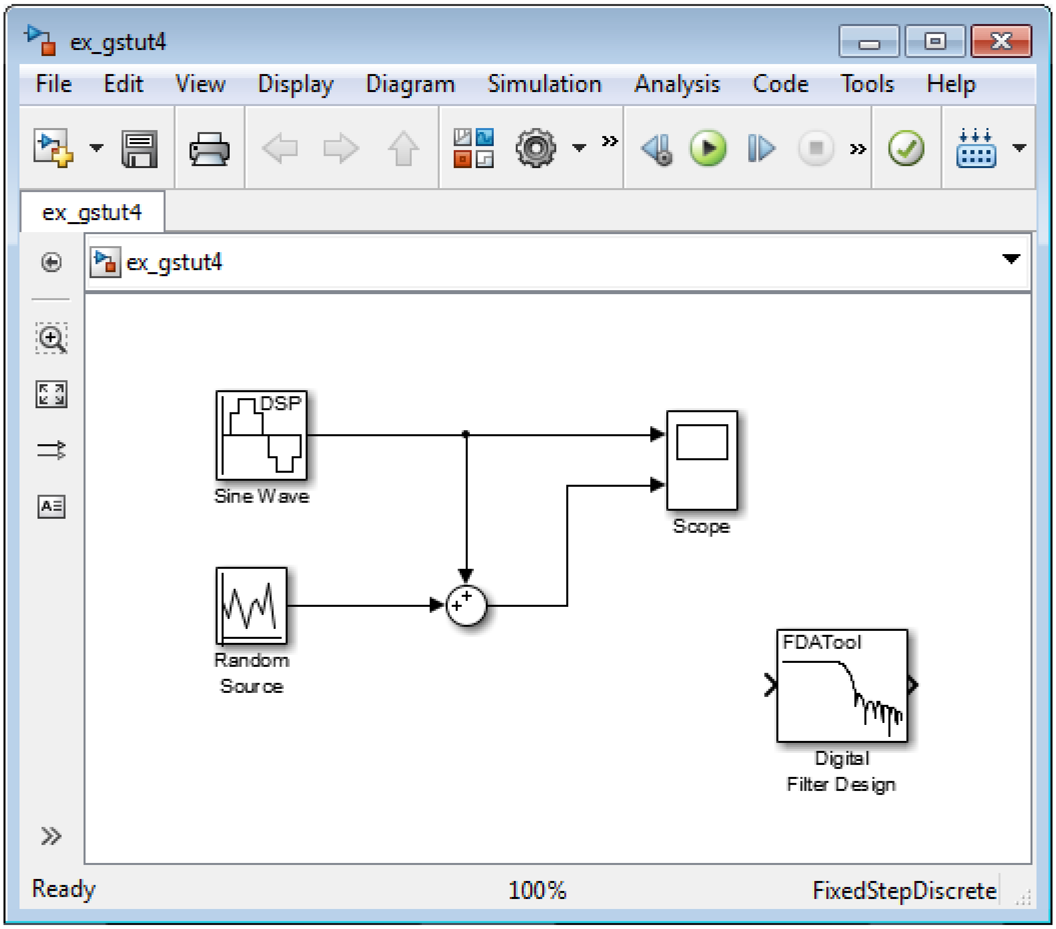
Add a Digital Filter to Your Model

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:

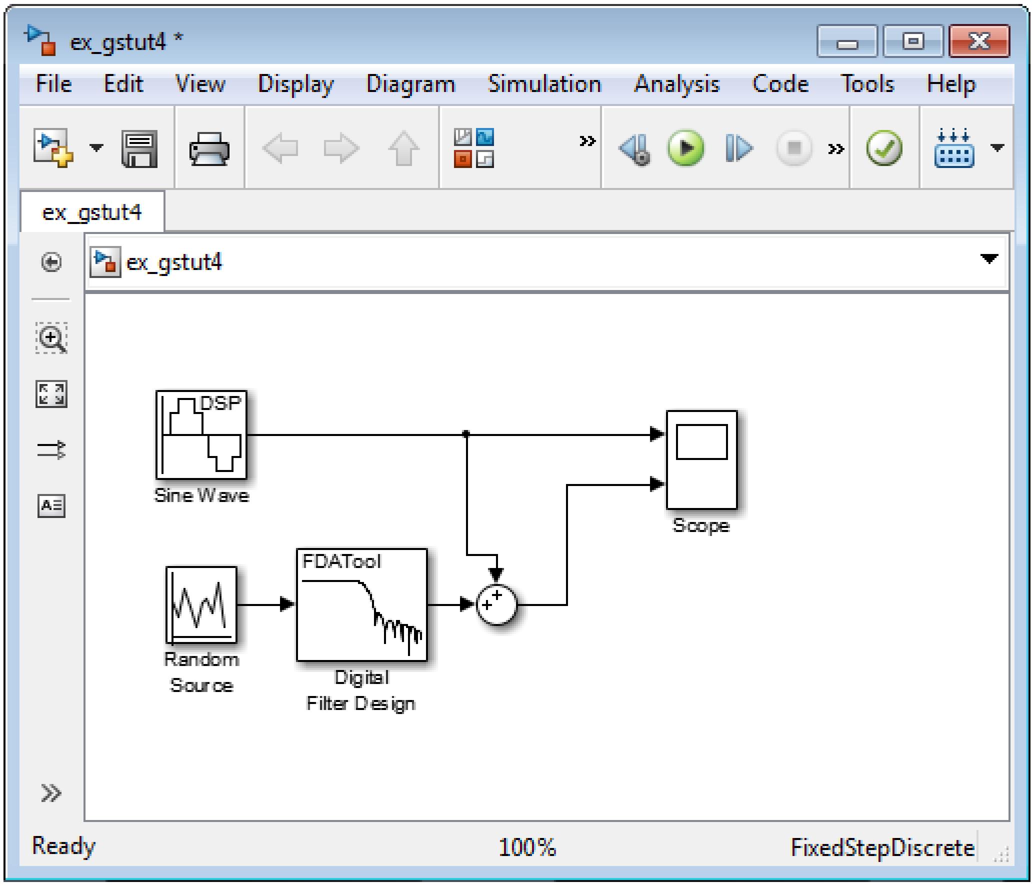
1. If the model you created in “Design a Digital Filter in Simulink” on page 3-2 is not open on your desktop, you can open an equivalent model by typing

   ```
   ex_gstut4
   ```

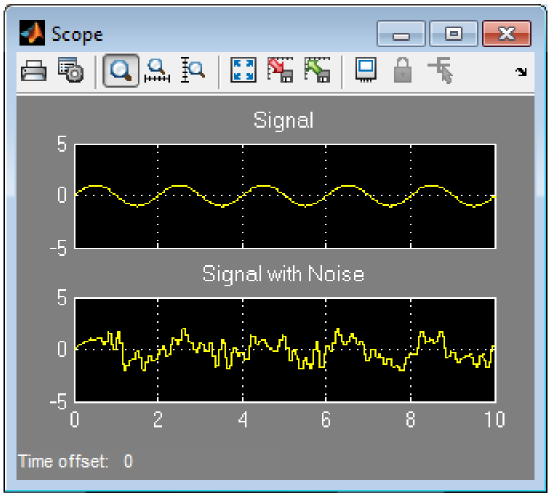
   at the MATLAB command prompt.



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

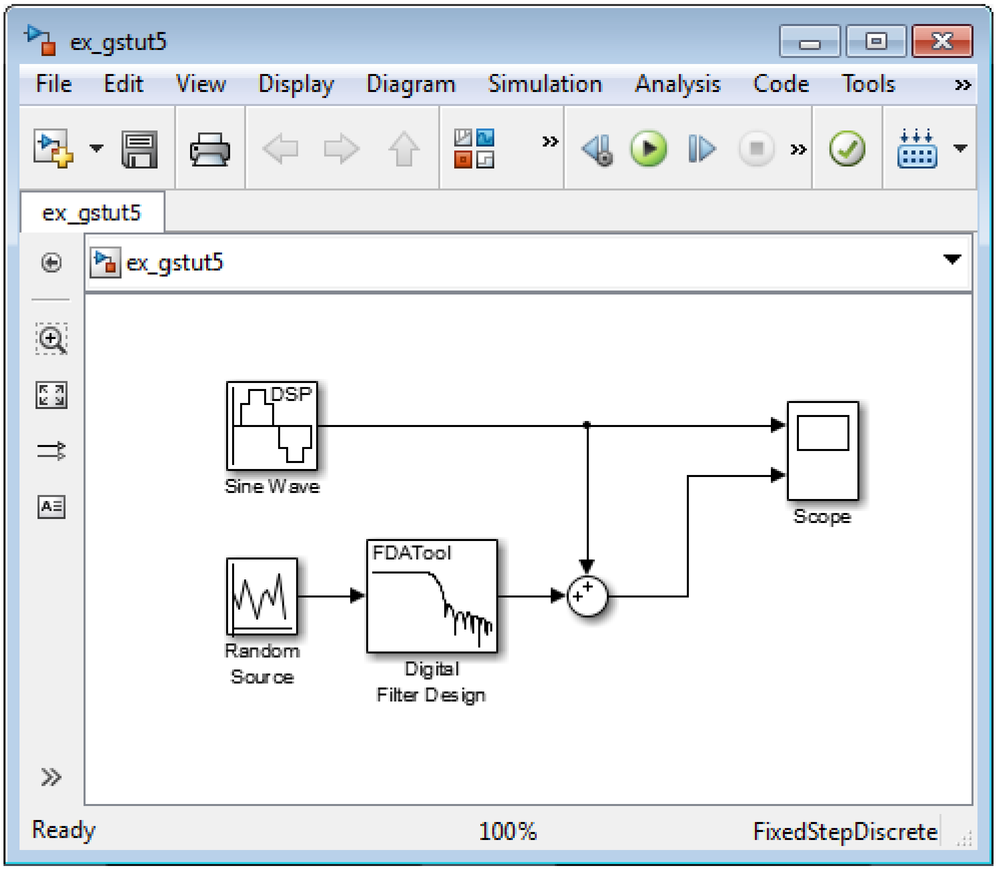
Adaptive filters track the dynamic nature of a system and allow you to eliminate time-varying signals. The DSP System Toolbox libraries contain blocks that implement least-mean-square (LMS), Block LMS, Fast Block LMS, and recursive least squares (RLS) adaptive filter algorithms. These filters minimize 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

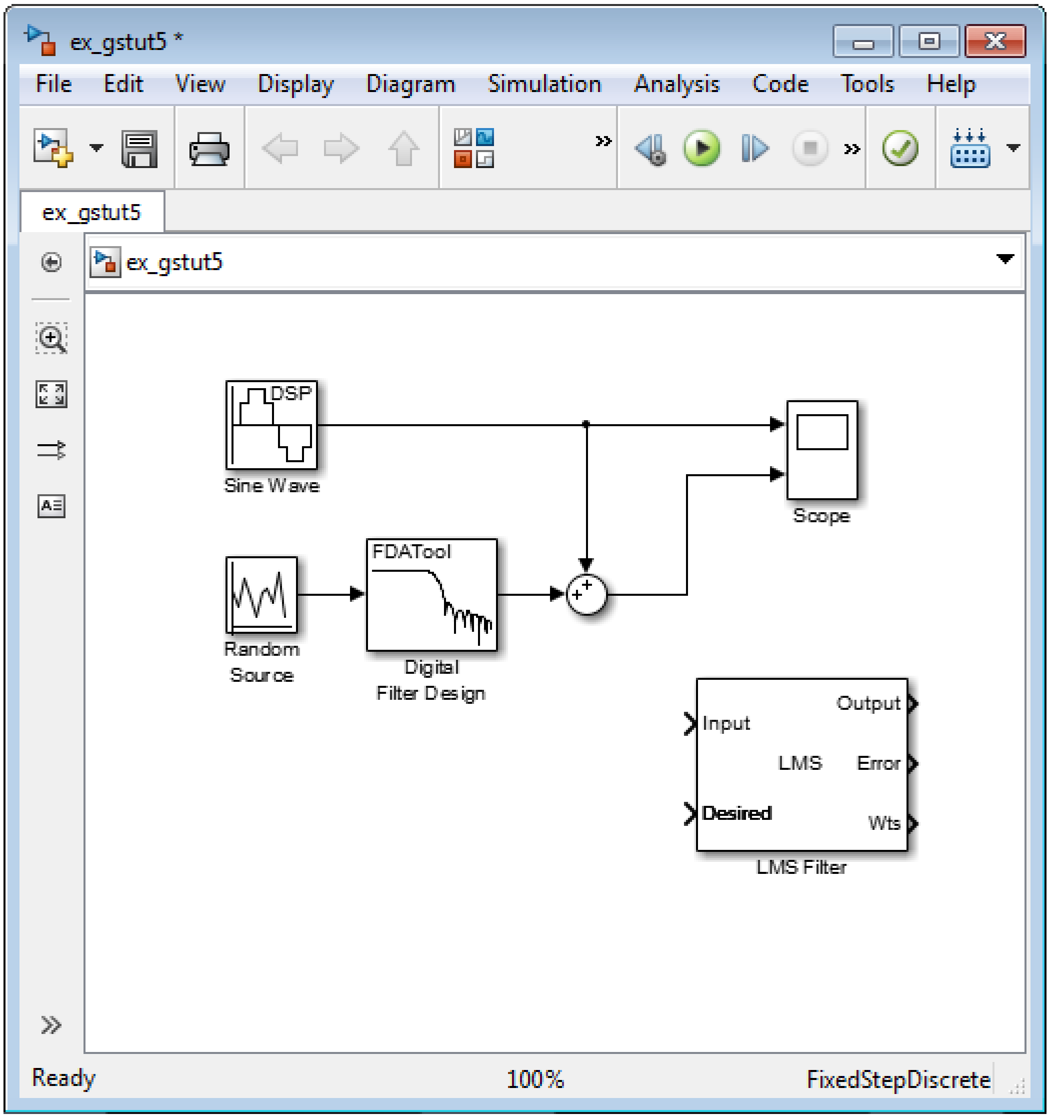
   ```
   ex_gstut5
   ```

   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.

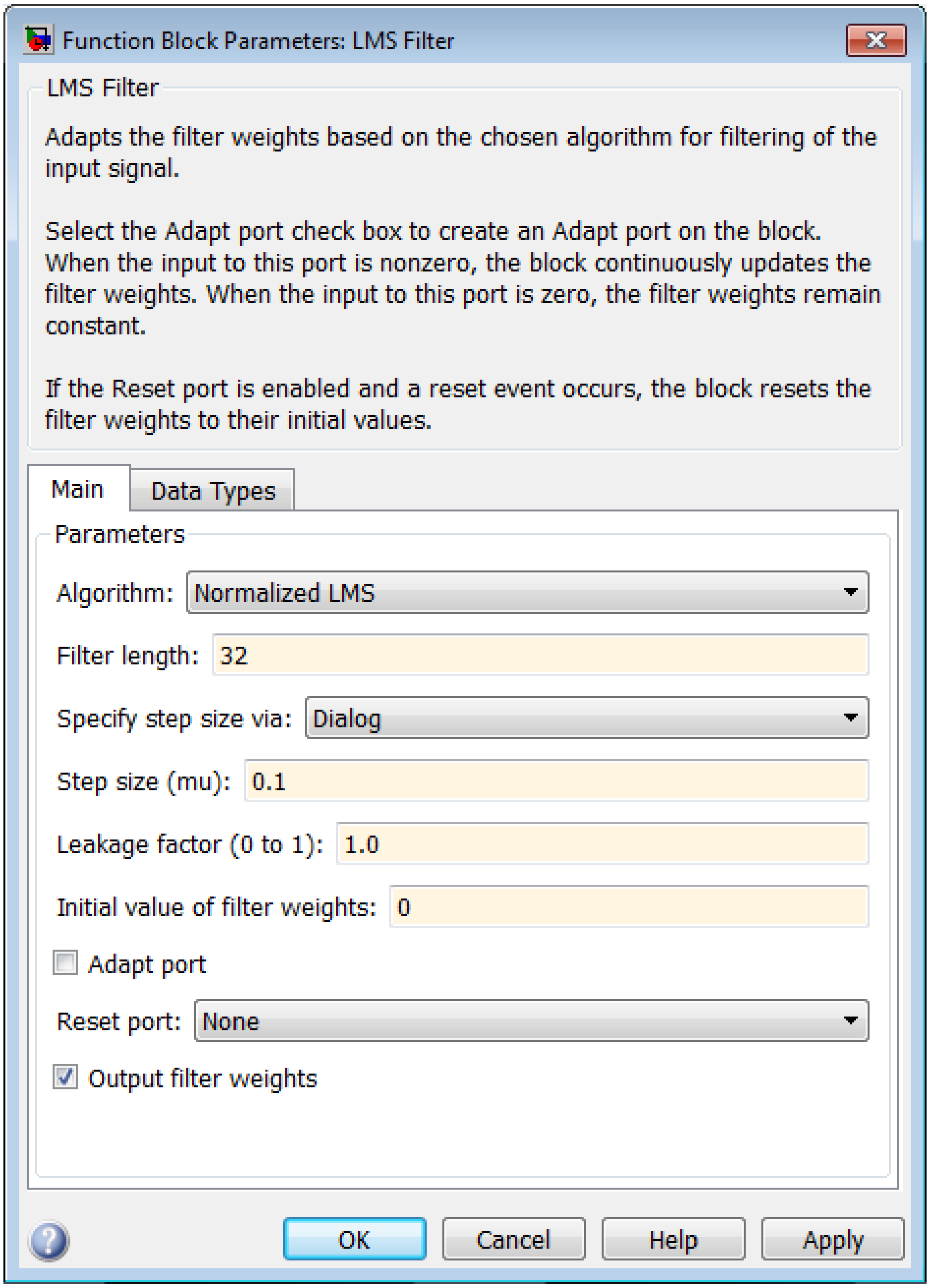


4 Set the LMS Filter block parameters to model the output of the Digital Filter Design block. Open its dialog box by double-clicking the block. Set the block parameters as follows:

- **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

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.

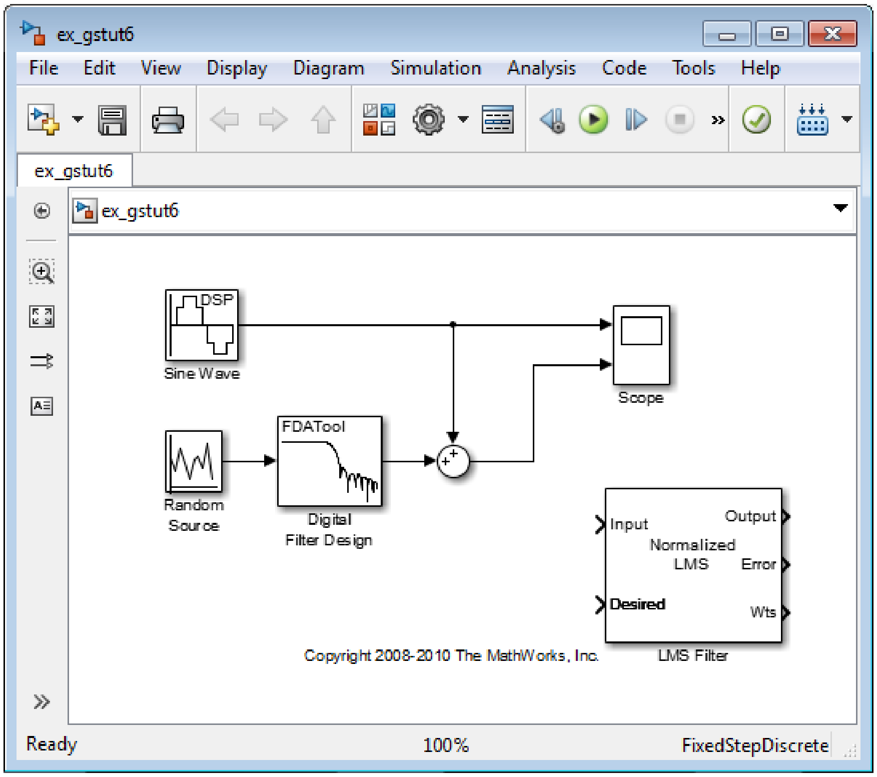
Add an Adaptive Filter to Your Model

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**.

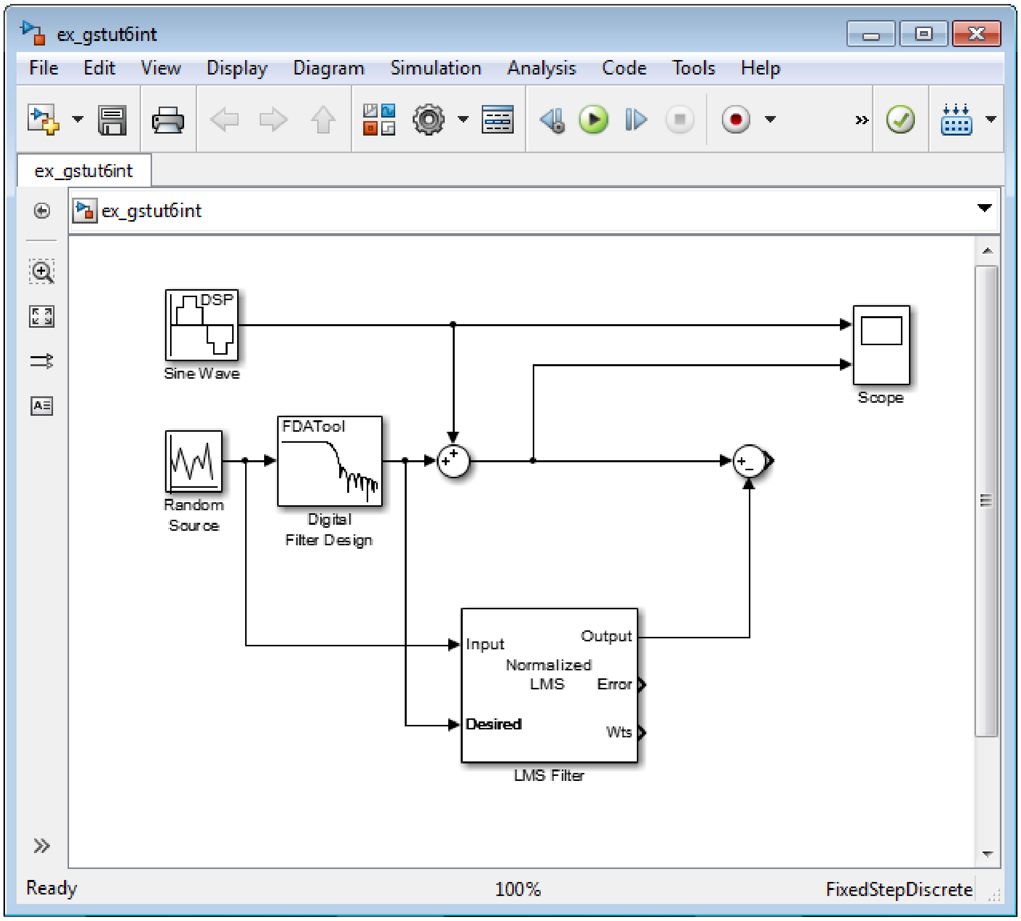
3 Incorporate the LMS Filter block into your system.

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

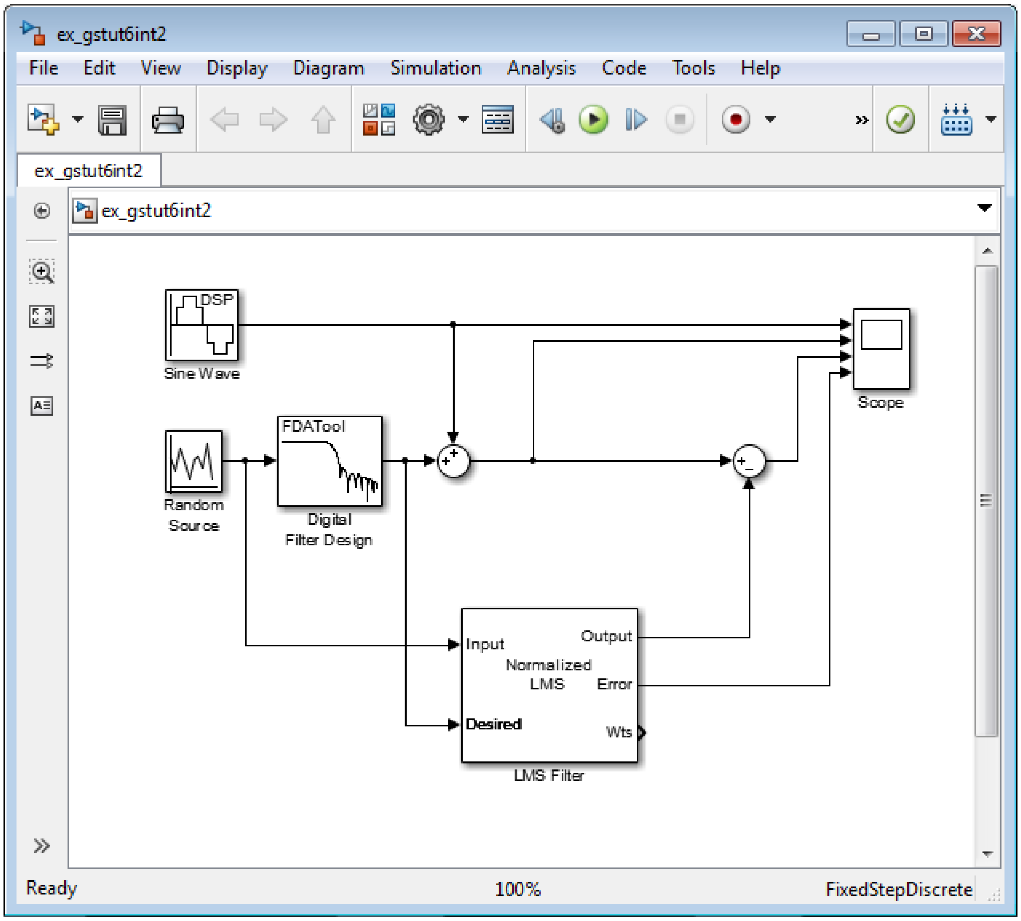
Because the LMS Filter block can only approximate the noise, there is still a difference between the input signal and the approximation of the input signal. In subsequent steps, you set up the Scope block so you can compare the original sinusoidal signal with its approximation.

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.

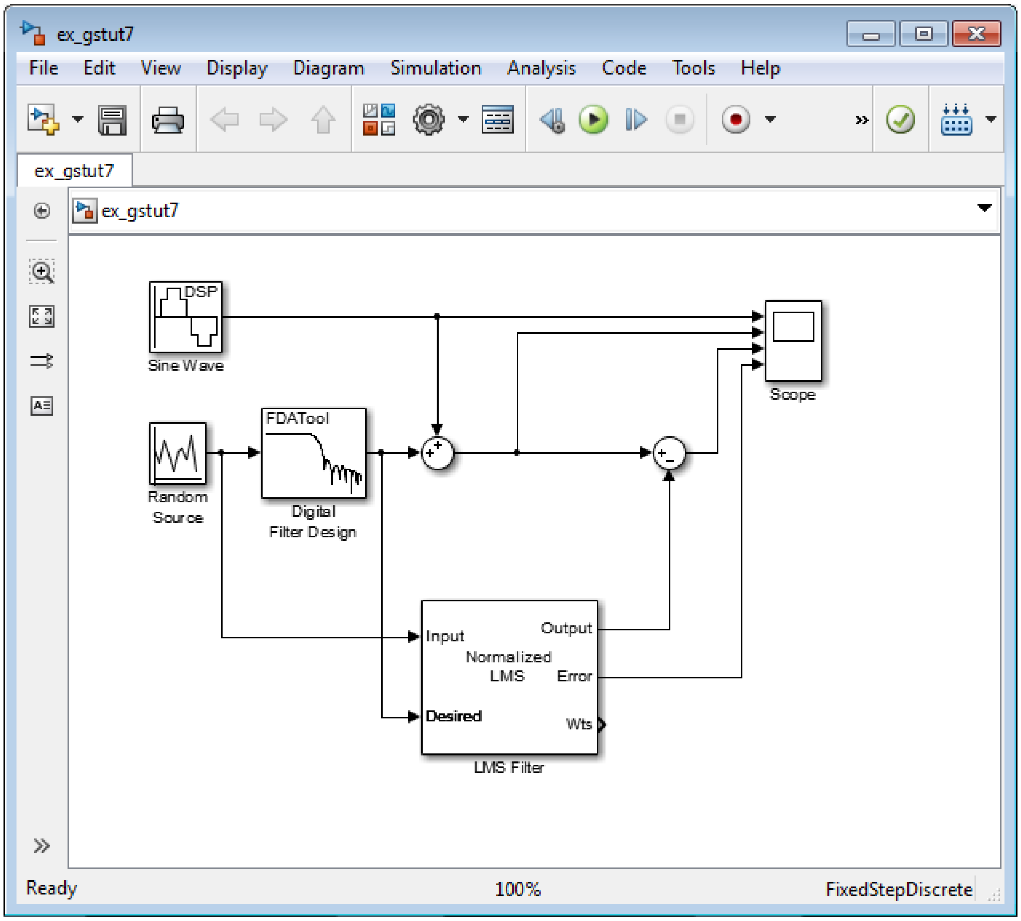
View the Coefficients of Your Adaptive Filter

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:

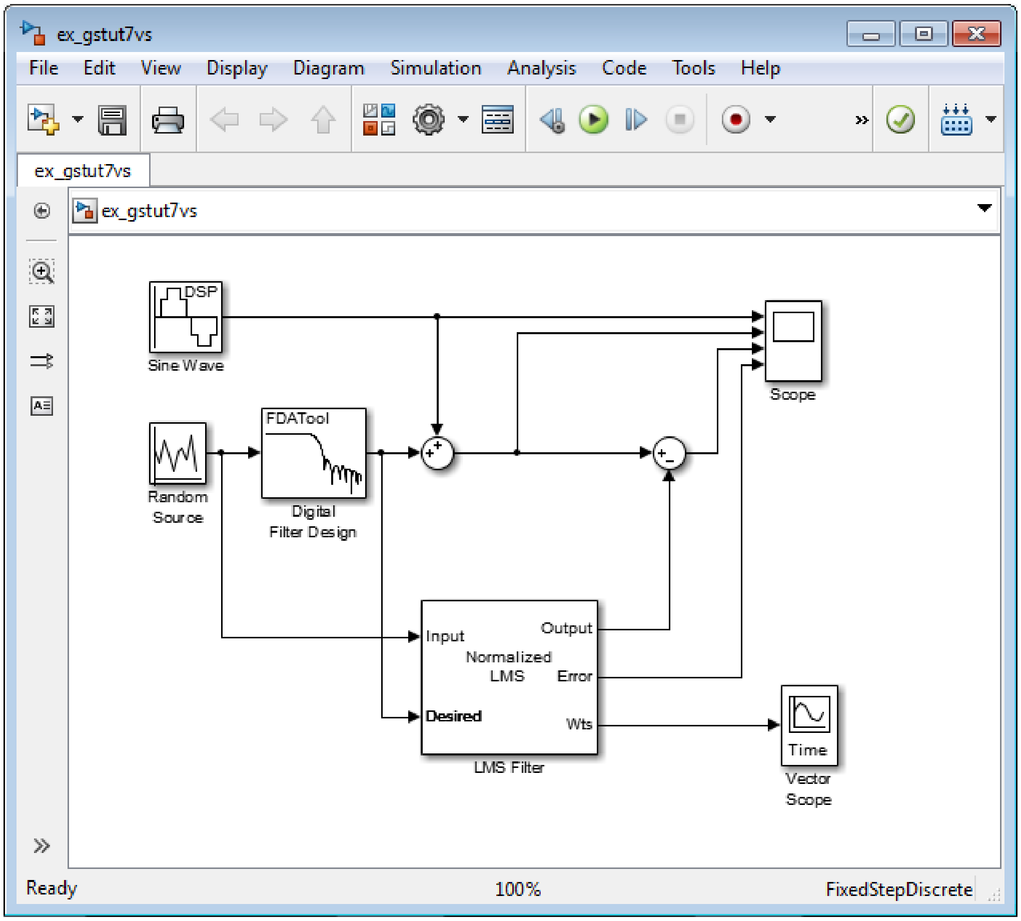
a Click the **Scope Properties** tab.

- **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.

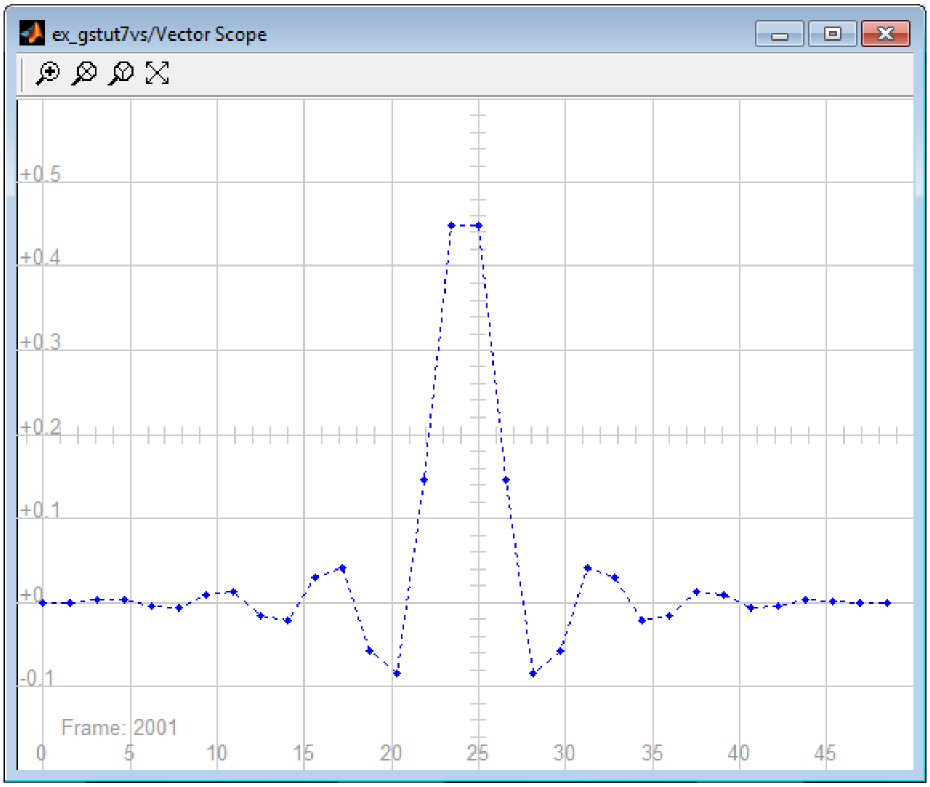
Note When working with models that contain DSP System Toolbox blocks, use source blocks that enable you to specify their sample time. If your source block does not have a **Sample time** parameter, you must add a Zero-Order Hold block in your model and use it to specify the sample time. For more information, 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.

---

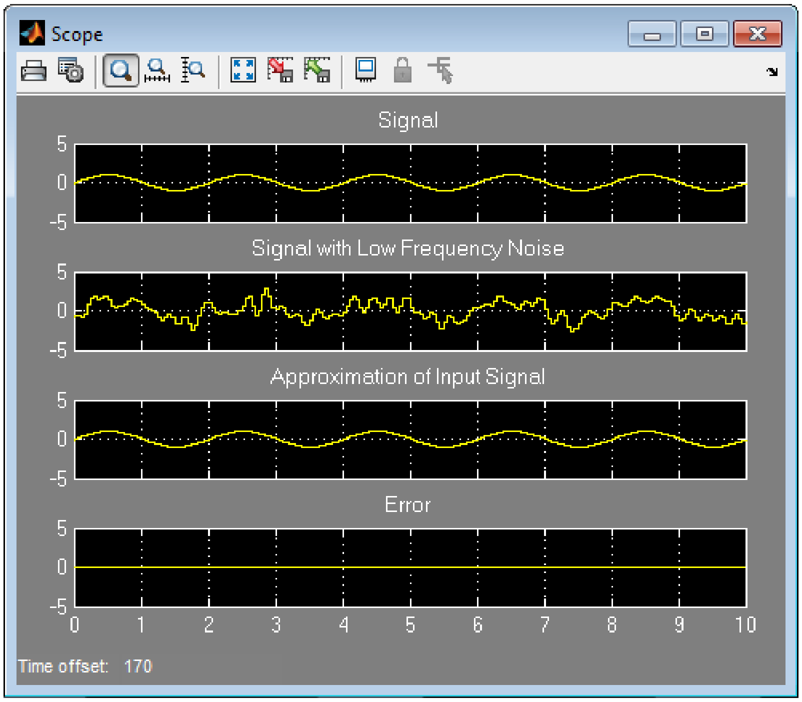
7. Close the dialog box by clicking **OK**.

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.


Scope
Signal
Signal with Low Frequency Noise
Approximation of Input Signal
Error
5
0
-5
0
1
2
3
4
5
6
7
8
9
10
Time offset: 170

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.

System Objects

Create System Objects

In this section...

A System object is aMATLAB object-oriented implementation of an algorithm. System objects extend MATLAB by enabling you to model dynamic systems represented by time-varying algorithms. System objects are well integrated into the MATLAB language, regardless of whether you are writing simple functions, working interactively in the command window, or creating large applications.

In contrast to MATLAB functions, System objects automatically manage state information, data indexing, and buffering, which is particularly useful for iterative computations or stream data processing. This enables efficient processing of long data sets.

System objects support fixed-point arithmetic. To use 64-bit data types, you must have Fixed-Point Toolbox™ software. System objects also support C-code generation from MATLAB and Simulink. With System objects, you can optionally generate code to target the desktop or external hardware. You can use System objects in Simulink® models via the MATLAB Function block. You can compile code that contains System objects within MATLAB functions using MATLAB Compiler™ software. (The compiler product does not support System objects in MATLAB scripts.)

Note System objects predefined in the software do not support sparse matrices. System objects you define support sparse matrices (see "Define a New System object" on page 4-3).

Create a System object

To use System objects, you must first create an object. For example,

```
H = dsp.FFT         % Create default FFT object, H

% Create input data
Fs = 1000;           % Sampling frequency
T = 1/Fs;            % Sample time
L = 1024;            % Length of signal
t = (0:L-1)*T        % Time vector

% Sum of two sinusoids
X = 0.7*sin(2*pi*50*t.') + sin(2*pi*120*t.');
```

Define a New System object

You can define a System object to implement your algorithm. For information and examples, see "Define New System Objects".

Change a System object Property

In general, you should set the object properties before you use the `step` method to run data through the object. To change the value of a property, use this format,

```
H.Normalize = true   % Set the Normalize property
```

The property values of the FFT object, `H`, are displayed.

Check if a System object Property Has Changed

To check if a tunable property has changed since `step` was last called, use this syntax:

```
flag = isChangedProperty(H,'Normalize')
```

`flag` is `true` if the `Normalize` property of object `H` has changed.

Run a System object

To execute a System object, use the `step` method.

```
Y = step(H,X);       % Process input data, X
```

The output data from the `step` method is stored in `Y`, which, in this case, is the FFT of `X`.

Display Available System Objects

To see a list of all the System objects for a particular package, type `help dsp`. To display help for specific objects, properties, or methods, see “Find Help and Examples for System Objects” on page 4-13 .

Set Up System Objects

In this section...

Create a New System object

You must create aSystem object before using it. You can create the object at the MATLAB command line or within a program file. Your command-line code and programs can pass MATLAB variables into and out of System objects.

For general information about working with MATLAB objects, see “Object-Oriented Programming” in the MATLAB documentation.

Retrieve System object Property Values

System objects have properties that configure the object. You use the default values or set each property to a specific value. The combination of a property and its value is referred to as a *Name-Value pair*. You can display the list of relevant property names and their current values for an object by using the object handle only, `<handleName>`. Some properties are relevant only when you set another property or properties to particular values. If a property is not relevant, it does not display.

To display a particular property value, use the handle of the created object followed by the property name: `<handle>.<Name>`.

Example

This example retrieves and displays the `TransferFunction` property value for the previously created `DigitalFilter` object:

```
H.TransferFunction
```

Set System object Property Values

You set the property values of a System object to model the desired algorithm.

Note When you use Name-Value pair syntax, the object sets property values in the order you list them. If you specify a dependent property value before its parent property, an error or warning may occur.

Set Properties for a New System object

To set a property when you first create the object, use Name-Value pair syntax. For properties that allow a specific set of string values, you can use tab completion to select from a list of valid values.

```
H1 = dsp.DigitalFilter('CoefficientsSource','Input port')
```

where

- `H1` is the handle to the object
- `dsp` is the package name.
- `DigitalFilter` is the object name.
- `CoefficientsSource` is the property name.
- `'Input port'` is the property value.

Set Properties for an Existing System object

To set a property after you have created an object, use either of the following syntaxes:

```
H1.CoefficientsSource = 'Property'
```

or

```
set(H1,'CoefficientsSource','Property')
```

Use Value-Only Inputs

Some object properties have no useful default values or must be specified every time you create an object. For these properties, you can specify only the value without specifying the corresponding property name. If you use value-only inputs, those inputs must be in a specific order, which is the same as the order in which the properties are displayed. Refer to the object reference page for details.

```
H2 = dsp.FIRDecimator(3,[1 .5 1])
```

specifies the `DecimationFactor` as `3` and the `Numerator` as `[1 .5 1]`.

Process Data Using System Objects

In this section...

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)`.

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 4-8.

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. See "Understand System object Modes" on page 4-11.
clone	Creates another object with the same property values
isLocked	Returns a logical value indicating whether the object is locked. See "Understand System object Modes" on page 4-11.
reset	Resets the internal states of the object to the initial values for that object
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.
isChangedProperty	Returns true if the specified tunable property value has changed since the last call to step.Example: flag = isChangedProperty(obj,'propertyName')
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.

Method	Description
`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.
`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.

Advantages of Using Methods

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.

Tuning System object Properties in MATLAB

In this section...

Understand System object Modes

System objects are in one of two modes: *unlocked* or *locked*. After you create an object and until it starts processing data, that object is in unlocked mode. You can change any of its properties as desired.

The object initializes and locks when it begins processing data. The typical way in which an object becomes locked is when the step method is called on that object. To determine if an object is locked, use the `isLocked` method. To unlock an object, use the `release` method. When the object is locked, you cannot change any of the following:

- Number of inputs or outputs
- Data type
- Dimensions of inputs or tunable properties, except for System objects that support 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 compile time and, therefore, cannot change at run time.
- Value of any nontunable property

Several System objects do not allow changing the complexity of inputs from real to complex. You can, however, change the input complexity from complex to real without unlocking the object.

These restrictions allow the object to maintain states and allocate memory appropriately.

Change Properties While Running System Objects

When an object is in locked mode, it is processing data and you can only change the values of properties that are *tunable*. To determine if a particular System object property is tunable, see the corresponding reference page or use a command of this form:

```
help dsp.FFT.Normalize
```

where

- `dsp` is the package name.
- `FFT` is the object name.
- `Normalize` is the property name.

Note Unless otherwise specified, System object properties are not tunable.

For information on locked and unlocked modes, see "Understand System object Modes" on page 4-11.

Change System object Input Complexity or Dimensions

During simulations you can change an input's complexity from complex to real, but not from real to complex. You cannot change any input complexity during code generation.

For objects that do not support variable-size input, if you change the input dimensions while the object is in locked mode, the object produces a warning and unlocks. The object then reinitializes the next time you call the step method. See the object's reference page for more information. You can change the value of a tunable property and the input size without a warning or error being produced. For all other changes at runtime, an error occurs.

Find Help and Examples for System Objects

Refer to the following resources for more information about System objects.

- Package help – `help dsp`, where `dsp` is a product package name
- Object help – `help dsp.FFT`, where `FFT` is the object name
- Documentation reference pages for an object – `doc dsp.FFT`
- Property help — `help dsp.FFT.Normalize`, where `Normalize` is the property name.
- Fixed-point property help – `dsp.FFT.helpFixedPoint`, where `helpFixedPoint` is the standard way to get fixed point property information for any System object.
- Method help – `help dsp.FFT.step`, where `step` is the method name.

To view examples, go to the Help contents for the associated product. Under `Examples`, select `MATLAB Examples`.

Use System Objects in MATLAB Code Generation

In this section...

Considerations for Using System Objects in Generated Code

You can generate C/C++ code from System objects using MATLAB Coder product. Using this product with System objects, you can generate efficient and compact code for deployment in desktop and embedded systems and accelerate fixed-point algorithms. System objects also support code generation using the MATLAB Function block in Simulink and the MATLAB Coder `codegen` function.

For general information on generating code, see

- MATLAB Coder product
- Simulink Coder product
- Embedded Coder® product

The following example, which uses System objects, shows the key factors to consider, such as using persistent variables, passing property values, and extrinsic functions, when you make MATLAB code suitable for code generation.

```
function lmssystemidentification
% LMSSYSTEMIDENTIFICATION System identification using
% LMS adaptive filter
%#codegen

    % Declare System objects as persistent.
    persistent hlms hfilt;

    % 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);

        % Create system (an FIR filter) to be identified.

        hfilt = dsp.DigitalFilter(...
                    'TransferFunction', 'FIR (all zeros)', ...
                    'Numerator', fir1(10, .25));
    end

    x = randn(1000,1);                              % Input signal
    d = step(hfilt, x) + 0.01*randn(1000,1);    % Desired signal
    [~,~,w] = step(hlms, x, d);                     % Filter weights

    % Declare functions called into MATLAB that do not generate
    % code as extrinsic.

    coder.extrinsic('stem');

    stem([get(hfilt, 'Numerator').', w]);
end

% To compile this function use codegen lmssystemidentification.
% This produces a mex file with the same name in the current
% directory.
```

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

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

Usage Rules for System Objects in Generated MATLAB Code

- Assign System objects to `persistent` variables.
- Global variables are not supported. To avoid syncing global variables between a MEX file and the workspace, use a compiler options object. For example,

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

  Then, include `'-config f'` in your `codegen` command.
- Initialize System objects once by embedding the object handles in an `if` statement with a call to `isempty( )`.
- Call the constructor exactly once for each System object.
- Set arguments to System object constructors as compile-time constants.
- Use the object constructor to set System objectproperties because you cannot use dot notation for code generation. You can use the `get` method to display properties.
- Test your code in simulation before generating code.

Limitations on Using System Objects in Generated MATLAB Code

- Ensure that size, type and complexity of inputs do not change.
- Ensure that the value assigned to a nontunable or public property is a constant and that there is at most one assignment to that property (including the assignment in the constructor).
- For most System objects predefined in the software, the only time you can set their properties during code generation is when you construct the objects. System objects that support tunable properties at any time during

code generation are listed in the product's code generation support table. For System objects that you define, you can also change their tunable properties at any time during code generation.

- Do not change the size of properties during code generation.
- The only System objectmethods supported in code generation are
  - `get`
  - `getNumInputs`
  - `getNumOutputs`
  - `isDone` (for sources only)
  - `reset`
  - `step`
- Do not set System objects to become outputs from the MATLAB Function block.
- Do not pass a System objectas an example input argument to a function being compiled with `codegen`.
- Do not pass a System objectto functions declared as extrinsic (i.e., functions called in interpreted mode) using the `coder.extrinsic` function. Do not return System objects from any extrinsic functions.

Use System Objects with 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.

Use System Objects with the MATLAB Function Block

Using the MATLAB Function block, you can include a MATLAB language function in a Simulink model. This model can then generate embeddable code. You can include any System object in the MATLAB Function block. 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.

Use System Objects with MATLAB Compiler

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

Index